9-1-1999

Guaranteed bandwidth implementation of message passing interface on workstation clusters

Ali Atalay

GUARANTEED BANDWIDTH IMPLEMENTATION OF MESSAGE PASSING INTERFACE ON WORKSTATION CLUSTERS

By

Ali S. Atalay

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Graduate Advisor – Muhammad E. Shaaban, Assistant Professor

Roy Czernikowski, Professor

Tony H. Chang, Professor

Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
September 1999

Thesis Release Permission Form

Rochester Institute of Technology

College of Engineering

Title: Guaranteed Bandwidth Implementation of Message Passing Interface On Workstation Clusters.

I, Ali Serdar Atalay, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or in part for non-commercial uses.

Signature:_____

Date: 9/10/97

ABSTRACT

Due to their wide availability, networks of workstations (NOW) are an attractive platform for parallel processing. Parallel programming environments such as Parallel Virtual Machine (PVM), and Message Passing Interface (MPI) offer the user a convenient way to express parallel computing and communication for a network of workstations. Currently, a number of MPI implementations are available that offer low (average*) latency and high bandwidth environments to users by utilizing an efficient MPI library specification and high speed networks.

In addition to high bandwidth and low average latency requirements, mission critical distributed applications, audio/video communications require a completely different type of service, guaranteed bandwidth and worst case delays (worst case latency) to be guaranteed by underlying protocol. The hypothesis presented in this paper is that it is possible to provide an application a low level reliable transport protocol with performance and guaranteed bandwidth as close to the hardware on which it is executing.

The hypothesis is proven by designing and implementing a reliable high performance message passing protocol interface which also provides the guaranteed bandwidth to MPI and to mission critical distributed MPI applications. This protocol interface works with the Fiber Distributed Data Interface (FDDI) driver which has been designed and implemented for Performance Technology Inc. commercial high performance FDDI product, the Station Management Software 7.3, and the ADI / MPICH (Argonne National Laboratory and Mississippi State University's free MPI implementation).

---

* In this thesis, the latency refers to average time it takes to transfer a message from the moment the send operation starts to the time when the receiver receives the packet. Worst case latency is the longest time recorded from connection establishment to the connection termination.

Acknowledgements

I am indebted to Professor, Roy Czernikowksi, who helped me finishing writing of my thesis and the paper. I am grateful for his support. He reviewed my thesis many times, and corrected my grammatical errors.

I would like to thank to my advisor, Professor Shaaban, who was always available with his valuable comments and support. Special thanks to other thesis committee member, Professor Chang for reading my thesis. The courses I took from him helped me during the development of this thesis.

I owe a great depth to my previous boss, my forever friend, John Grana who assigned me all the toughest device driver projects in my junior years at Performance Technologies Inc. where I learned the UNIX device driver development, protocol internals with hands on assignments.

I am grateful to my beloved girlfriend Elcin. She was so kind and understanding. She visited me so many times while I can not afford to visit her even once during this thesis write-up.

Most of all, I would like to dedicate this thesis to the memory of my father, "Cengiz Atalay", to my mom, sister and to my beloved girlfriend Elcin.

Table of Contents

List of Figures

List of Tables

Glossary

ADI[20]:	Abstract Device Interface.
ATM[19]:	Asynchronous Transfer Mode.
DLPI[16]:	Unix Systems Laboratories Data Link Provider Interface specification.
DMA:	Direct Memory Access.
ESS:	End Station Service: Client to SBA.
FDDI[6]:	Fiber Distributed Data Interface.
IETF[37]:	Internet Engineering Task Force.
IP[35]:	Internet Protocol.
LAN:	Local Area Network.
LAM[17]:	Ohio Supercomputing Center MPI implementation.
LLC:	Logical Link Control.
MAC:	Media Access Control Sublayer.
MMAP:	Memory mapping.
MPI:	Message Passing Interface.
MPI-CH[2]:	Argonne MPI implementation.
MPI-FM[8]:	University of Illinois, Fast Messages implementation of MPI
MPI-GB:	Guaranteed bandwidth implementation of MPI.
MPI-RT[4]:	Real Time Message Passing Interface Specification.
MPP:	Massively Parallel Processor.
MTU[37]:	Maximmum Transmission Unit
NOW[34]:	Network of Workstations.
NIC:	Network Interface Card.
OSI[37]:	Open System Interconnection
P4[18]:	Parallel Programming System developed Argonne National Laboratory in 1990.
PHY[6]:	Physical Layer.
PMD[6]:	Physical Media Dependent.
PVM[1]:	Parallel Virtual Machine.

PTI:	Performance Technologies Inc.
RTP[37]:	Real Time Protocol.
QoS[38]:	Quality of Service.
SBA[6,7]:	Synchronous Bandwidth Allocation.
SNAP:	Sub-Network Access Protocol.
SMT[6]:	ANSI FDDI Station Management Software.
SU[6,7]:	Synchronous Units
TCP[35]:	Transport Control Protocol.
TTCP[35]:	Benchmarking tool, used as a load injector in this work.
TLB:	Translation Lookaside Buffer.
UDP[37]:	User Datagram Protocol
WAN:	Wide Area Network.
XTP[36]:	Xpress Transfer Protocol.

1. Introduction

1.1 General

Workstation clusters are increasingly being used as cost-effective parallel computing platforms. These parallel programming environments offer the user a convenient way to express parallel computation and communication.

A number of libraries such as PVM [1], MPI [2] are already available to provide parallel computing environments on a workstation clusters. MPI [2] is a library specification for message passing, proposed as a standard by a broadly based committee of vendors, implementers and users. This standard and its extensions, MPI II and Draft MPI/RT [3-4], feature a range of functionality, including point-to-point communication with synchronous and asynchronous communication modes, and collective communication. MPI basically is designed to give MPI implementers freedom in choosing exactly how and when data is moved from one process to another.

Among MPI implementations, MPICH [5], developed by Argonne National Laboratory, stands out as portable, and allows higher performance than other available implementations. MPICH is built on a lower level communication layer called Abstract Data Interface (ADI). ADI defines low-level communication-related functions that can be implemented in different ways on different machines. This interface is designed to support multiple devices (e.g., TCP/IP, and shared memory) in a single MPI application. Abstract Data Interface (ADI) contains routines for packetizing the messages and attaching the header information, managing multiple buffering policies, matching posted receive requests with incoming messages or queuing them if necessary. The upper layers

of MPICH handles the rest of the MPI standard, including the management of data types, and communicators, and the implementation of collective operations with point-to-point operations. (Figure 10). More information regarding MPICH and ADI can be found in chapter 2. The device implementations are typically built on top of the TCP/IP protocol stack because of its wide usage.

The performance offered by TCP/IP and consequently delivered to the libraries is not acceptable since both the protocol itself and its traditional implementations within the operating system were designed for objectives like tolerance to high latency and high error rates rather than low latency and high bandwidth. These communication protocols require a number of services to work, like timers, buffer management, process protection, and process notification. Besides, these protocol stacks come as a part of the operating system, require a message copy and do not allow network interface cards to perform direct memory access (DMA) into their buffer areas.

An efficient low level messaging protocol should be designed to provide a low-level reliable transport protocol with point-to-point and multicast capabilities and with performance as close as possible to the hardware on which it is running on.

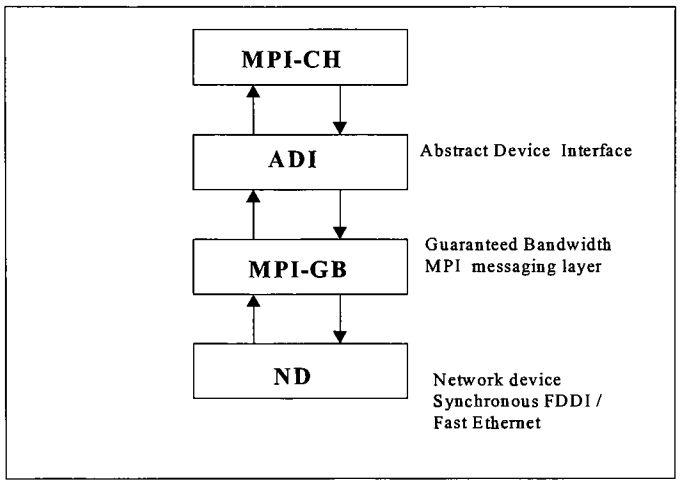
It has been observed that the message layers incur significant software overheads when implementing "high level" features not provided by hardware. These software overheads are due to both the network hardware (no flow control, no multicast) and to poor performing device drivers (not thread-safe, over mutex locks, no interrupt-based message delivery, redundant data copying). Low latency, low host utilization and high

performance are very important characteristics for the underlying network to ensure an efficient message passing among workstation clusters.

When designing today's mission critical communication protocols, it is also very important to guarantee the deadlines of synchronous messages while sustaining a high aggregate throughput. Synchronous FDDI [6](Fiber Distributed Data Interface) provides guaranteed bandwidth and a bounded access time for synchronous messages and high bandwidth (100 Mbit/sec) to the network interface.

The goal of this thesis is to design a reliable high performance message passing protocol abstract data interface which will provide the guaranteed bandwidth to MPI and to the Quality of Service (QoS) required by mission-critical distributed MPI applications. The intention is to bring the performance and guaranteed bandwidth to user applications as much as possible. MPI-GB (Guaranteed Bandwidth for MPI) runs on top of the FDDI driver designed and implemented for Performance Technology's Inc. commercial high performance FDDI product. This product utilizes the SMT 7.3 standard SBA [7] (Synchronous Bandwidth Allocation) and delivers predictable performance to the network interface. The protocol software (MPI-GB) interfaces with ADI/MPICH (Argonne National Laboratory and Mississippi State University's free MPI implementation) (Figure 1) and with the FDDI device driver to provide this guaranteed bandwidth to user applications.

Figure 1: MPI-GB Model



1.2 Motivation

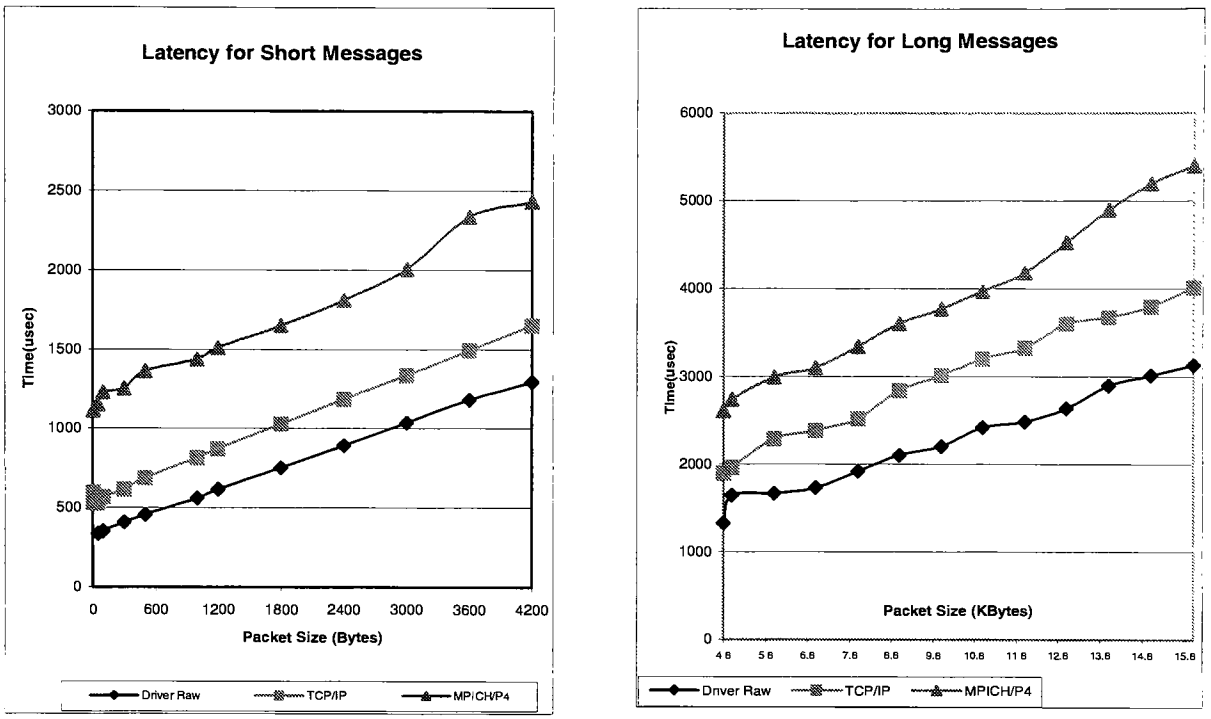
In the initial phase of this thesis, various a number of interface products for several commercial and non-commercial operating systems are benchmarked. The benchmarking suites included various tests to measure a combination of network latency, bandwidth, and contention. The performance results, given on the following pages, provide a good insight into the behavior of architectures, device drivers, and protocols under different load conditions.

Typically, less than half of the performance* provided by the network interface is delivered to user applications (MPI) for most packet sizes. The results in figures 2 and 3 are recorded using PCI SPARC workstations. (The characteristics of these workstation and others used in this thesis is given in Table 12.) This ratio drops even lower while

* Average bits/sec and latency delivered to application vs the max bandwidth and min. latency can be delivered from FDDI to device drivers. First Results are always ignored in latency and bandwidth computations. (See Appendix for the test source code)

running at "Slow Host & Fast Network" test setups. (Please refer to chapter 2 and 4 for more information regarding the discussion regarding these benchmarks).

Figure 2: *Ping – Pong latency for Raw, TCP/IP and MPI APP.*



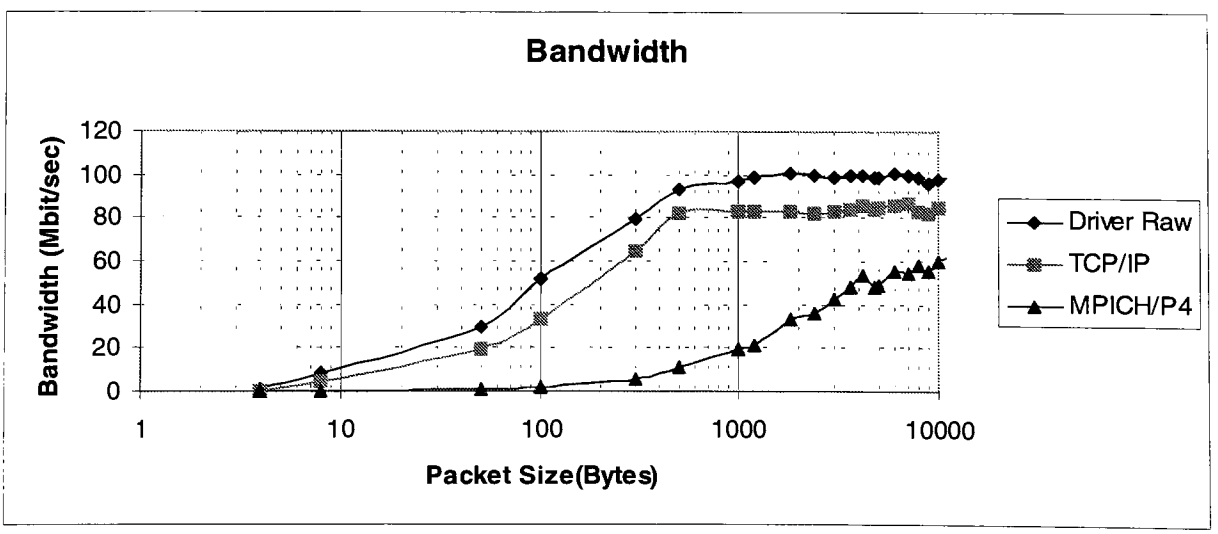


Figure 3: *Bandwidth for Raw, TCP/IP and MPI APPS.*

But, "Where does the time go?" as Dr.Chien[11] asked for MPPs.

To find where the time is wasted, different OSI protocols TCP/IP, MPI level protocols that interfaces with network interface architecture through the device driver and our the raw driver data transfer have been benchmarked. Carefully analyzing the results helped in identifying the following problems and bottlenecks in software where this time is lost:

- Poor network device drivers and architectures

  Non interrupt-based message delivery or interrupting more than necessary, lack of DMA or overusing DMA (ex. even for small messages, not establishing a balance with Processor I/O and DMA)

  Not thread safe device drivers, over mutex locking.

  Redundant data moving

  Non-streamlined, non-optimized code increases instruction cache misses.

  No support for synchronous traffic and collision (Ethernet)

  Slow I/O Buses (Sbus)

- Heavyweight, over-reliable, slow transport protocols like TCP/IP.

  Designed to achieve reliable and efficient networking over slow, unreliable networks. It meets its goal by timers, buffers, process protection, three-way handshake at the expense of loosing valuable time. Additional process overhead and extra copying in and out from the communication buffers needs to removed.

  No support for prioritized or real-time delivery.

- Message passing interface (MPI) protocol overheads

  Protocol overhead related with MPI itself and p4 abstract data device.

  Extra data copying from MPI/P4 to BSD Socket interface.

- Overheads related with communication between these distinct layers.

  Non-streamlined interfaces and distinct layering as a result of that the upper layer's information is not available to lower layer

  Redundant Context switching for system calls and no real-time scheduling.

After determining the problem areas, the next step is to design a protocol that would eliminate these overheads, get MPI closer to hardware and try to bring the performance and average latency to the level provided by hardware.

A number of research groups have already worked on improving communication latency and bandwidth by modifying message passing protocols to facilitate efficient system implementation. Some examples include the Fast Message [8] project by Chien, Shrimp project by Li [9], and URTP by Bruck [10]. More detail about these others can be found in chapter 2.

New communication technologies (ATM[19], FDDI[6], Fiber channel[28]) bring a completely different kind of service and guaranteed bandwidth to the network interface. FDDI uses a separate synchronous channel other than a regular asynchronous channel for stream-oriented traffic such as audio and video that require a low latency, and high bandwidth pipe. Like other new technologies, FDDI provides guaranteed bandwidth to the network interface only. User applications can benefit from this technology only with

carefully designed protocols. By designing and implementing MPI-GB protocol we show that it is possible to get the user applications benefit from these features the network interface provides, without loosing much in reliability.

1.3 Deliverables

The purpose of this thesis is to show that it is possible to bring the features of a network interface like high performance, low average and worst-case latency and guaranteed bandwidth to a distributed user application. To serve this purpose, existing programming environments, standards, protocols and recent research are investigated. Upon this investigation, MPI-GB protocol is designed and implemented to meet the requirements needs of mission-critical distributed applications. In the design, two main concepts are implemented: 1) Being given a very reliable underlying network, extra error checking should be removed from the protocol to achieve a minimal protocol overhead. 2) Using real-time schedulers, priority queuing, and removing extra copying, the gap between the user application and network hardware should be narrowed.

These two concepts are demonstrated by the implementation of the MPI-GB protocol. MPI-GB is benchmarked and compared to other protocols using average and worst-case latency, average and guaranteed bandwidth.

1.4 Methodology

To achieve the goals given in this thesis, various layers of the software such as the device driver, network interface API, MPI and its abstract data interface are studied individually and all together. The extra protocol overhead, buffering and inefficiencies arising at the

interface between these layers on receive and transmit sites are removed. Special care is given for scheduling between layers and passing data between layers. Finally static buffering and flow control is carefully integrated into MPI-GB for a reliable and efficient data transfer. For example if the maximum threshold value is reached on the receive site, the receive site sends a control packet to the sender to slow down. Using this negative acknowledgement approach helps in eliminating the overhead caused by acknowledgment packets [*]. Early threshold warning approach gives the sender application an option to take precautions such as slowing down the transmission, therefore allowing the receiver a chance to process and free more buffers before all receive buffers are depleted.

(Please refer to section 3.1 of this thesis for detailed explanation of the design methodology).

1.5 Results

During high and idle asynchronous load, MPI-GB and other MPI implementations are evaluated based on generic network evaluation metrics such as bandwidth, Ping-Pong (Roundtrip) latency and guaranteed bandwidth metrics such as worst case roundtrip latency to Average latency ratio and percentage of packets missing their deadlines.

Our results indicate that MPI-GB provides much better roundtrip latency than MPI-CHP4 during idle load. The roundtrip latency results of MPI-GB versions present an incremental improvement over the previous versions. All three implementations outperform the MPI-CHP4 implementation for all message sizes.

[*] Many techniques have been developed to decrease the number of acknowledgments for transport protocols. The Examples are adjusting the acknowledgment frequency with window based flow control for TCP [38], Selective retransmission for XTP [39].

For short messages, MPIGB-DLPI and MPIGB-MMAP roundtrip latency results are very close to raw data transfer results obtained from the device driver. The difference between raw data transfer and MPI implementations for the long packets can be attributed to the extra copy done to satisfy the hardware's DMA engine requirements.

MPI-GB primarily serves for guaranteed message delivery. Worst-case roundtrip message delivery latency is nearly one and half times of the average roundtrip message latency during high load. This ratio was anywhere between 7 to 22 for MPICH-P4.

GB, itself is evaluated against UDP/IP when an external load is injected by TTCP. Along with RTP, UDP can be considered as a real time protocol. The GB message miss ratio is only about 5 % during high asynchronous network loads (50 Mbit/sec), whereas this value is close to 76 % for UDP/IP packets during this load.

Inconsistent results and failures are recorded in back-to-back send packets bandwidth test due to the lack of optimizations on the rate and flow control of the GB protocol. This is one of the areas where MPI-GB needs to be improved.

1.6 Thesis Overview

The outline of the thesis is as follows:

In chapter 2, we provide more background information on the hardware and software used in this thesis. This section discusses the paradigm of parallel processing on networks of workstations (NOW). It continues with a brief overview of FDDI and its OSI model. It elaborates on synchronous FDDI device driver design and implementation and how FDDI fits in to the MPI-GB model. The remainder of this section gives background

information of parallel programming APIs such as Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) available on NOW. It continues with introducing and comparing MPI implementations: MPI-CH and Ohio Supercomputer Center's MPI implementation (LAM). It classifies the existing high performance message passing systems and compares them with each other and MPI-GB briefly.

Chapter 3 discusses MPI-GB and its building blocks in detail. This chapter starts with explaining MPI-GB's abstract device interface and continues with discussing the general design methodology, identifies the problem areas and concentrates on how it is solved in three different MPI-GB versions.

In Chapter 4, the performance measurements of three versions of MPI-GB are presented along with analysis of results. Metrics such as average and worst case latency, average and guaranteed bandwidth are used in the analysis. This chapter compares MPI-GB with the others using the metrics listed above.

Finally in Chapter 5, the most important lessons we learned during the design and implementation of MPI-GB are discussed. This section also lists the accomplishments and future improvements that can be made to MPI-GB.

2. Background

2.1 Network of Workstations (NOW)

As the performance to price ratio of desktop workstations continues to increase every year, they are becoming common parallel computing platforms to solve high-end scientific and engineering problems. These networks provide wiring flexibility, scalability, and incremental expansion capability. Utilizing load balancing techniques and using widely available software on these networks allow us to distribute some computation of heavily loaded workstations to the idle or lightly loaded workstations. This helps to prevent the waste of potential useful resources and degradation of the overall response time.

Researchers and developers of these platforms are addressing many different problems to meet the requirements of highly demanding distributed applications. The first problem is the communication overheads. Since the existing communication technologies such as LAN and WAN were not initially developed for parallel processing, the communication overheads among the workstations are very high. To solve this bottleneck, the researchers proposed two different solutions. One solution is the use of existing fast interconnection technologies such as FDDI, Gigabit Ethernet, and ATM. The other solution is to use custom interconnects such as U-Net [33], Myrinet [8].

Solving the communication overhead problem is important but not sufficient. In order to bring MPP-like performance and reliability, NOW should have high performance messaging protocols, parallel programming environments supporting load balancing, scheduling, and fault tolerance. In this thesis, we intend to design and implement a high performance messaging protocol to contribute to a solution of this problem.

As an interconnection network, due to the lack of custom interconnect networks like Myrinet, we chose FDDI because of its availability, reliability and synchronous bandwidth support.

Figure 4: *TCP/IP bandwidth for Sbus and PCI Workstation pairs*

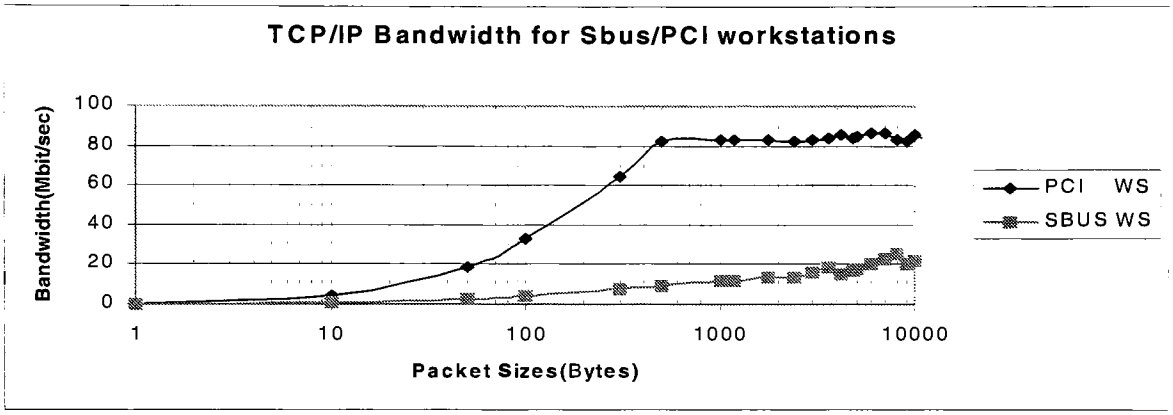
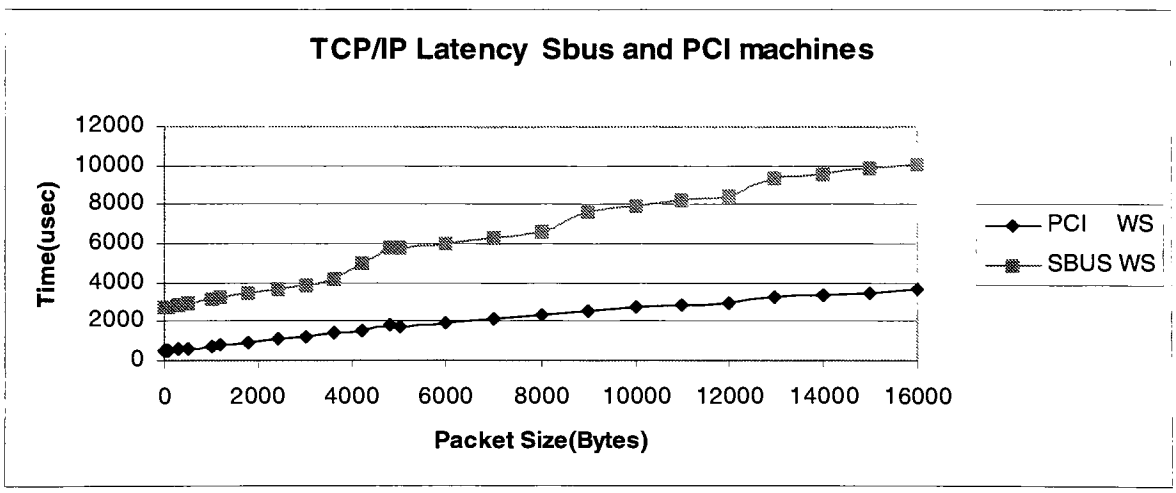


Figure 5: *TCP/IP latency for Sbus and PCI Workstation pairs*



The figures above show the latency and bandwidth results for TCP/IP packets for various packet lengths recorded using Sbus and PCI SPARC workstation pairs with FDDI network listed in Table 1 (PTI Test Bed). The major difference in two samples is not only due to the different CPU speeds, cache sizes and I/O buses (PCI vs SBus). Network protocols like TCP/IP assume that the network is so slow **relative** to the host CPU and

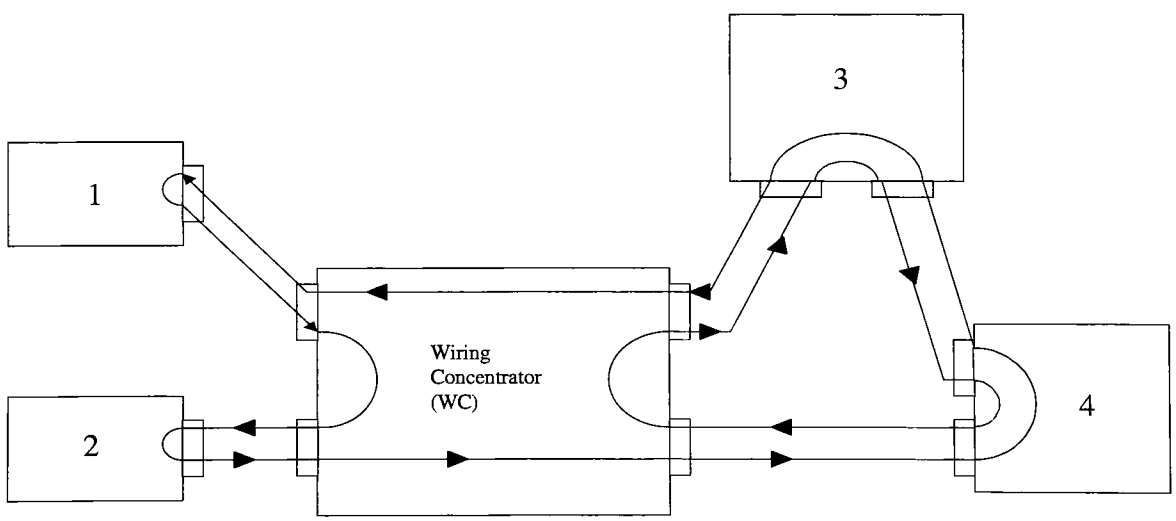
memory system that it can afford to use poorly optimized, general purpose data structures and paths. FDDI could be considered as a fast network for old SBus SPARC machines. The overhead added by network protocols by data copying in and out of the user buffers and network buffers can be tolerated on fast hosts but not on slower hosts.

The next section briefly overviews FDDI and its OSI model. It elaborates on synchronous device driver design and implementation and how FDDI fits in to the MPI-GB model.

2.2 Fiber Distributed Data Interface (FDDI) and its interface to MPIGB

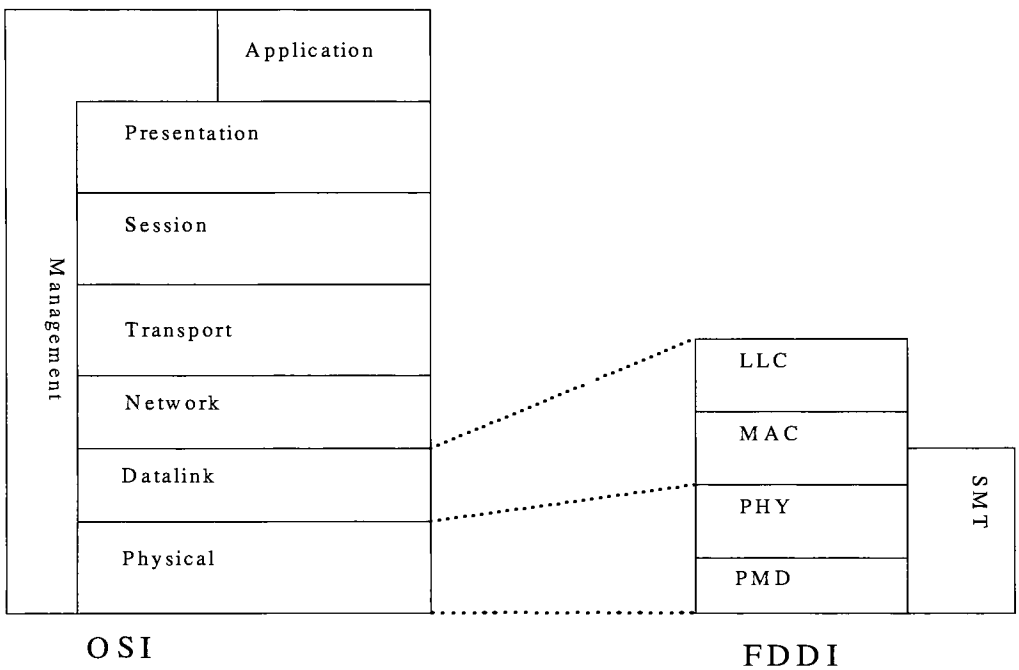
FDDI is a 100 Mbps data rate local area network standard. Figure 6 shows a typical FDDI network.

Figure 6: *Typical FDDI Topology*



Some stations are configured on FDDI with two rings and some are configured with one. The dual attachment stations (DASs) use two fibers, and the single attachment stations (SASs) use one fiber. The IEEE 802.1 standard specifies the generic OSI reference model for LAN. FDDI network model is compared with the generic OSI model in Figure 7.

Figure 7: *OSI and FDDI model*



The FDDI physical layer is divided into two sublayers: (a) the physical layer (PHY) protocol and (b) the physical layer medium dependent (PMD) interface. The PMD interface is responsible for defining the transmitting and receiving signals, providing proper power levels, and specifying cables and connections. The physical layer protocol is intended to be medium independent. It defines symbols, coding and decoding techniques, clocking requirements, the states of lines, data framing conventions.

The data link layer consists of logical link control (LLC) and media access control (MAC). The FDDI MAC provides the procedures used for frame formatting, error checking, token handling, multiple classes of traffic, dual mode of operation and managing the data link addressing.

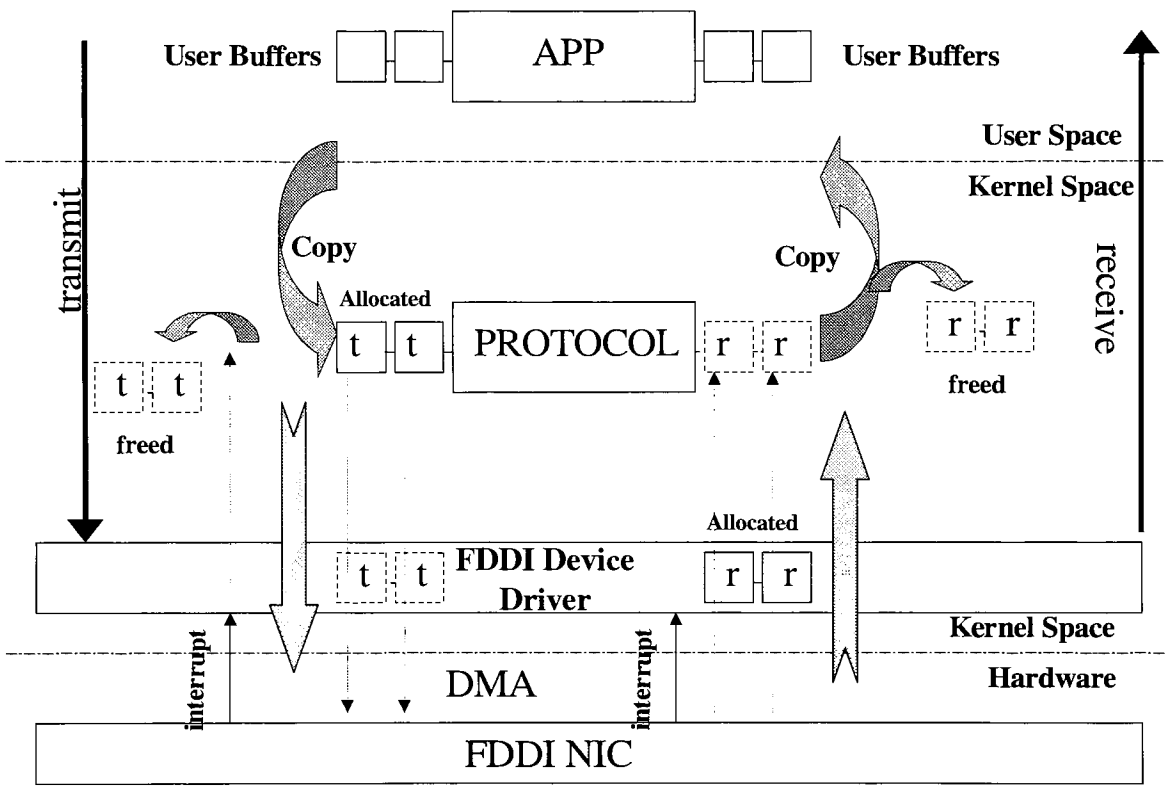
FDDI also has a station management (SMT) function. The station management standard provides the procedures for managing the station attached to FDDI. It provides procedures for remote/local node configuration, error statistics, error detection and recovery, and connection management.

As mentioned above, FDDI MAC protocol provides multiple classes of service: synchronous and asynchronous. The synchronous class of service has a guaranteed bandwidth and access delay. It is also the highest priority traffic. This is very useful for applications that require deterministic access to the network, minimum jitter (variations in inter-frame arrival times in destination), bandwidth guarantee requirements. This synchronous class is necessary for stream-oriented traffic such as voice, video and mission critical distributed control applications, which require a deterministic response time. The asynchronous class of service on the other hand, is the remaining bandwidth, which is dynamically shared between all nodes.

For the initial phase of this thesis, a Solaris OS™ device driver for PTI's FDDI network interface card is designed and implemented. The host-NIC interaction is achieved by utilizing the NIC's three DMA channels. The host message buffer memory is directly memory mapped to the board, and DMA transfers to/from the host are performed through this kernel memory. No buffer copying is performed inside the device driver; the physical

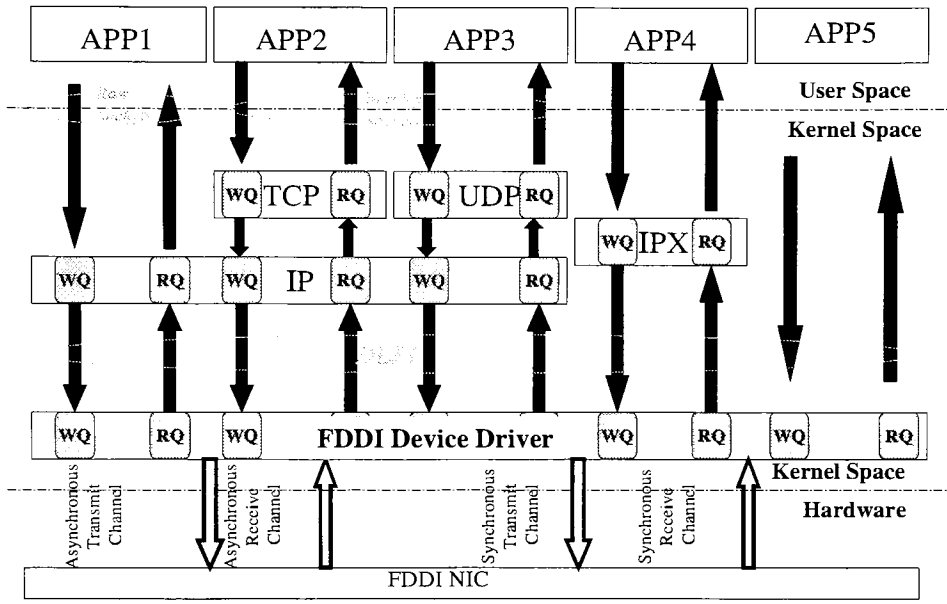
pointer address of the message buffers coming from the transmit path are directly given to the board for DMA transfer. The physical pointers of the pre-allocated message buffers are provided to the board's DMA receive channel for the incoming packets.(Figure 8)

Figure 8: *Buffer management in FDDI device driver*



The driver is fully Data Link Provider Interface Specification compatible; raw and IP applications talk to the device driver via DLPI commands utilizing Solaris OS™ streams. (Figure 9)

Figure 9: FDDI Device Driver software layers



In the first and second versions of MPI-GB implementation, this FDDI driver is used without any modifications. In these versions, MPI-GB resides in user space and utilizes DLPI commands to communicate with the device driver. The first version, Raw DLPI implementation, MPI-GB opens the raw FDDI device and attaches and binds to this device with DLPI commands. The second version, Raw IP socket interface, MPI-GB opens an IP socket and talks to a device driver using this socket interface and IP layer. In both of these approaches, Solaris OS™ streams copies the user buffers to the kernel's message buffer area before giving the data to the device driver. As explained in previous chapters, this user to kernel buffer copying is one of the most important contributors to the overhead and should be avoided where possible. Also experiments show that, Solaris OS™ streams scheduler is not designed for real time scheduling. As documented in

Solaris man pages [22] the quanta vary from 20 ms to 200 ms; this of course can increase latencies in MPI-GB protocol and nullify all the good work that has been done.

To avoid this and to prevent the to/from kernel space data copying, the third version of MPI-GB, mmap user interface, has been designed. In this version, the device driver mmaps all transmit and receive kernel buffers to user space where MPI-GB can read/write from/to. To solve the scheduling problem, Sobalvarro's approach is used. [23] Raising the sleeping MPI-GB's process priority to maximum lets an MPI-GB process schedule and run immediately on a Solaris platform. Please refer to chapter 3 for the design details of the three approaches.

2.3 Parallel Development Tools

A number of parallel development tools have been used to support NOW such as p4[18], Parallel Virtual Machine (PVM) [1], and Message Passing Interface [2], There are currently two MPI implementations available. These are the Argonne implementation (MPI-CH) and the Ohio state implementation (LAM). We will briefly discuss these in the next section by emphasizing the argument that these libraries fail to bring the hardware's performance to the application.

2.3.1 PVM

The development of PVM started in 1989 to design a virtual machine consisting of a set of heterogeneous hosts connected to a network that appears logically to the user as a single large parallel computer [1].

It took researchers three years to accomplish this goal. PVM 3.0 was released in 1993 to enable a PVM application to run across a virtual machine composed of multiple large microprocessors. As the designers of PVM admit in [25], portability was considered much more important than performance. The researchers focused on problems with scaling, fault tolerance, and heterogeneity of the virtual machine rather than the performance, based on the fact that communication over the Internet was slow. Later research was focused on the performance problem. Subramaniam found in [26] that in many cases the PVM communication library achieves only 15-20% of the network's theoretical capacity. He determined that extra latency is partially due to overheads involved in TCP/IP communication and complex buffering scheme in PVM. He and Von Eicken [27] even mentioned in this writing a user-controlled device driver for the network interface. He thought that this involves extensive patches to the kernel. However MPI-GB is designed with no modification to the kernel except for the FDDI device driver.

The PVM message passing system consists of a daemon process and a set of communication primitives. PVM provides the standard message passing routines such as pvm_send(), which is a blocking send, and pvm_recv(), which is a blocking receive. Tasks communicate with each other using UDP sockets. A daemon process runs on each PVM machine. The daemons communicate among themselves to perform operations such as starting up a user task, multicasting messages, and finding the status of a particular task on a particular host. More information about PVM can be found in [1].

2.3.2 MPI

The acronym "MPI" stands for " Message Passing Interface". It refers to a combination of application programming interfaces and required behaviors that represent layers five through seven of the ISO OSI reference model for Networking. MPI-1 and MPI-2 are both parallel computing MPI standards.

A broadly based committee of vendors, implementers and users defined MPI-1 in 1993. The impetus for developing MPI was that each Massively Parallel Processor (MPP), workstation vendor was creating its own proprietary message passing API. In this scenario it was not possible to write a portable parallel application. MPI is intended to be a standard message-passing specification that each MPP, workstation vendor would implement on its system. One major goal of the MPI is to provide such a specification without sacrificing performance.

MPI contains a range of features including:

The ability to specify communication topologies,

-A large set of point-to-point and communication routines,

A communication context that provides support for the design of safe parallel software libraries,

The ability to create derived data types that describe messages of noncontiguous data.

In 1995 the MPI committee began meeting to design MPI-2 [3] specification to correct the ability to dynamically start MPI tasks on separate hosts and to add additional communication functions including:

-MPI_SPAWN functions to start both MPI and non-MPI process,

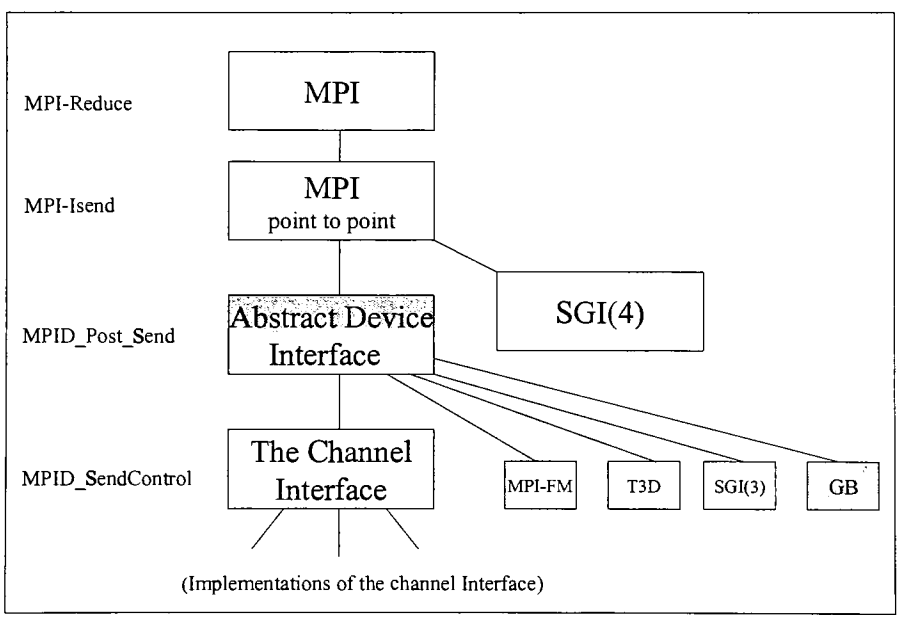
-One-sided communication functions such as put and get

-2 Nonblocking collective communication functions.

-Language bindings for C ++

Finally, MPI started meetings MPI/RT[4](Real Time extensions to MPI) in 1998. The MPI-RT would be a specification to offer extremely low cost of portability as compared to any native software architecture for messaging, while providing useful real-time notions of performance, predictability, and quality of service. During the writing of this thesis, the 8/98 draft of this standard is available at [4] and no implementation of this draft standard (MPI/RT) is available.

Figure 10: *MPICH layers.*



MPI is the only standard upon which many free and commercial implementations exist. The ones that have been used in this thesis are: MPICH, a portable implementation of MPI developed jointly by Argonne National Laboratory and Mississippi State University

and Ohio State University's MPI-2 implementation (LAM). As an interesting variation to commercial implementations, MPI Software Technology Inc is currently designing and implementing a Java based MPI (JMPI) [28].

2.3.2.1 MPICH

MPICH, developed by Argonne National Laboratory is a popular, free and a portable implementation of MPI. This implementation of MPI has been designed to simplify the task of porting MPI to new platforms by providing a multi-level implementation. The MPI-CH layers are shown in Figure 10.

The MPICH implementation contains an abstract device interface (ADI) that performs four main functions: sending, receiving, data transfer queuing, and device dependent functions. ADI contains nearly a dozen core routines [Table 1]. Implementing ADI routines is all that is required to port MPICH to a new platform [29]

***Table 1:** ADI core routines*

Function	EXPLANATION
MPID_Cancel	Cancel a pending open.
MPID_Check_device	Check for pending order
MPID_Complete_send	Complete a send
MPID_Complete_recv	Complete a receive
MPID_End	Terminate the ADI
MPID_Init	Initialize the ADI
MPID_Iprobe	Check if specific message has arrived.
MPID_Myrank	Rank of calling process
MPID_Mysize	Number of processes
MPID_Post_send	Start a send operation
MPID_Post_send_ready	Start a send, ready mode
MPID_Post_send_sync	Start a send, synchronous mode
MPID_Post_receive	Starts a receive operation
MPID_Test_send	Test for completion of a send
MPID_Test_recv	Test for a completion of a receive

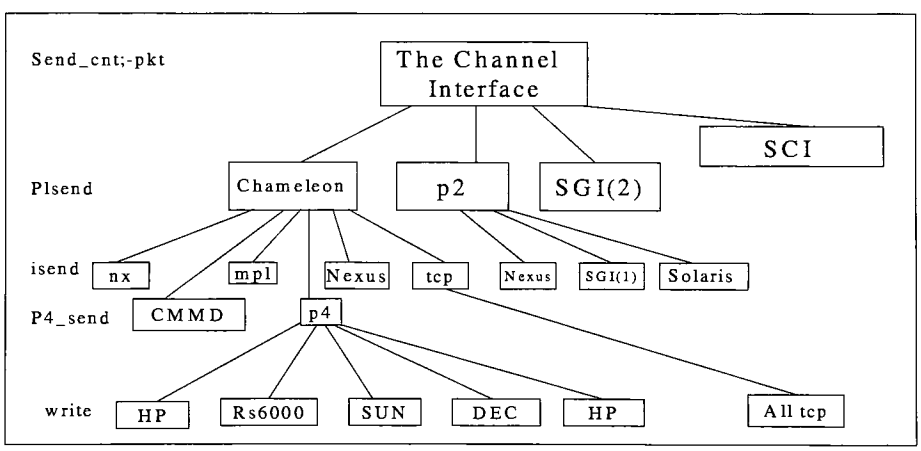
Keeping portability in mind, the MPICH implementers offer an additional portable layer, the channel interface, that contains only five routines (bare minimum) needed just for data transfer [Table 2].

Table 2: *Channel Interface Routines*

Function	EXPLANATION
MPID_Control_Msg_Available	Indicate whether a control message is available
MPID_RecvAnyControl	Read next control message. Block if none avail.
MPID_SendControl	Send a control message
MPID_RecvFromChannel	Receive a data from particular channel
MPID_SendChannel	Send data to a particular channel

MPICH includes multiple implementations of the channel interface.

Figure 11: *Channel Interface*



These implementations are chameleon [30], shared memory (p2), and architecture specific ones like SGI, and HP-Convex and SCI. Chameleon is the only implementation we found that could be used in the NOW paradigm. By implementing the channel interface in terms of Chameleon macros, the implementers provide the portability to a

number of systems, with no additional overhead, since Chameleon macros are resolved at compile time. The Chameleon implementation involves p4 [18] parallel programming system for Workstations to ensure portability to MPPs with the expense of bringing extra layers of overhead to overall performance. MPICH implementers claim that they are working on the TCP/IP and UDP/IP versions of the Chameleon interface and include TCP in Figure 11; at the time of this writing, these versions are not available [5].

The channel interface is completely ignored in our MPI-GB implementation. ADI primitives are directly implemented for performance reasons

With the help of this multi-layer approach, MPICH is currently available for a wide range of hardware platforms, from NOWs to MPPs like IBM SP, Ncube, Paragon, Meiko, Cray T3D. However, this level of portability comes at the expense of added software overhead and performance degradation.

The performance of raw data transfer, TCP/IP sockets and the MPICH-P4 implementation of MPI on FDDI networks is based on latency and bandwidth metrics. These test results are taken with the network setup detailed in Chapter 4. The performance comparison of raw data transfer for TCP/IP and MPI, gives a good estimate of the overheads associated with using higher level abstraction of protocol stacks.

Figure 2 and 3 show the latency and bandwidth of MPICH, TCP/IP and raw data transfer for different packet sizes. We proposed MPI-GB to improve the performance of MPI based on latency and bandwidth. The goal was to bring MPI data transfer performance close to raw data transfer performance levels. With the true zero copy mmap design and

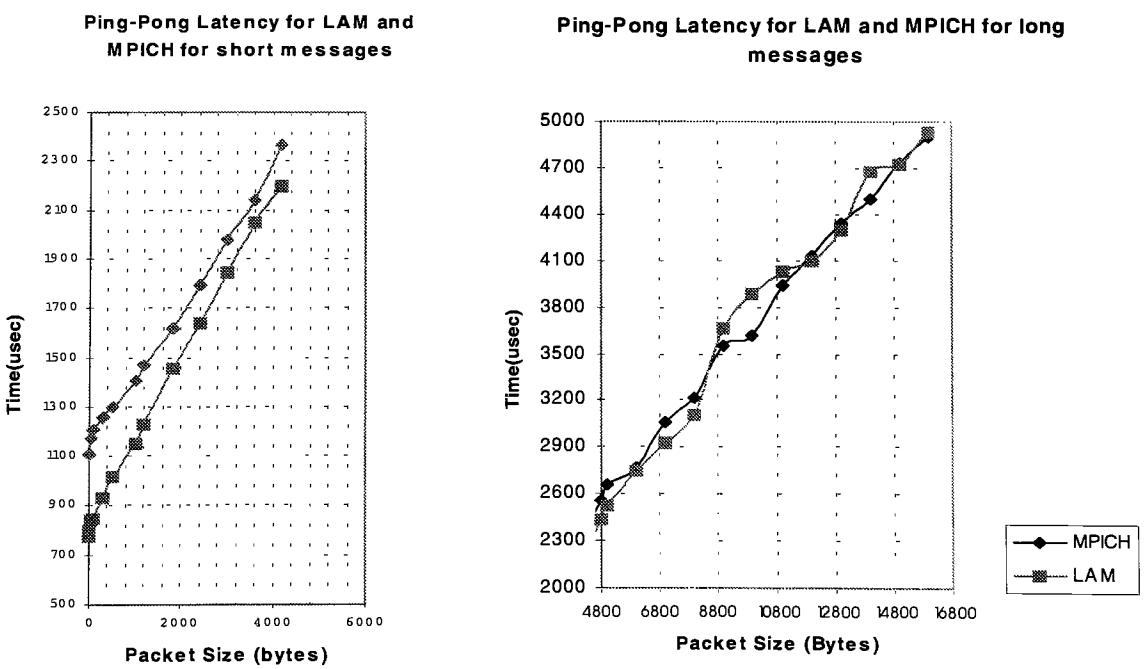
proper scheduling and implementation, the MPI-GB performs close to the raw data transfer rate and much better than TCP/IP for larger packets. (Refer to Section 4, figure 31)

2.3.2.2 LAM

LAM is another implementation of MPI developed at the Ohio Supercomputer Center of the Ohio State University. It is based on a network of daemon farms providing communication as well as remote program execution and file access. Unlike MPICH, LAM is based on the MPI-2 standard and provides features like dynamic process spawning. LAM and MPICH also differ in how they set up communication channels between MPI processes. MPICH makes connection on a demand driven basis whereas LAM sets up a fully connected network at initialization time.

The latency of LAM and MPICH was measured and compared by using the MPI ping latency program on SPARC PCI workstation pairs in the PTI test bed 1(Table 11). Figure 12 contains the results.

Figure 12: *Latency for MPICH and LAM*



All LAM tests use "–c2c", "-nger" and "–O" switches to mpirun. The first selects client-to-client mode in which the LAM library bypasses the daemon and clients communicate directly. The second turns off the Guaranteed Envelope Resources feature of LAM. The third informs the LAM/MPI library that the cluster is homogeneous and hence turns off the data conversion.

Obviously, LAM has a smaller latency in sending small packets less than 8Kbytes. When the message size is above 8Kbytes, the sending latency of LAM increased. This is due to the fact that LAM switches over to long message protocol for messages longer than 8192 bytes in length. By default MPICH changes protocol at 16384 bytes.

Please refer to Chapter 3 for design details of three different versions of MPIGB and Chapter 4 for the analysis and comparison of MPI-GB with LAM , MPICH and others.

2.4 MPI-GB and Related work

There have been a number of research efforts to improve the performance of message passing system in a NOW environment. The techniques for improving performance of message passing systems can be largely classified in Figure 13 [12]. We will focus on the MPI-FM [8], U-Net[33], fbufs[31], NCS HPI[12] and Fast Sockets implementations. These implementations provided a valuable framework while designing and developing the MPI-GB.

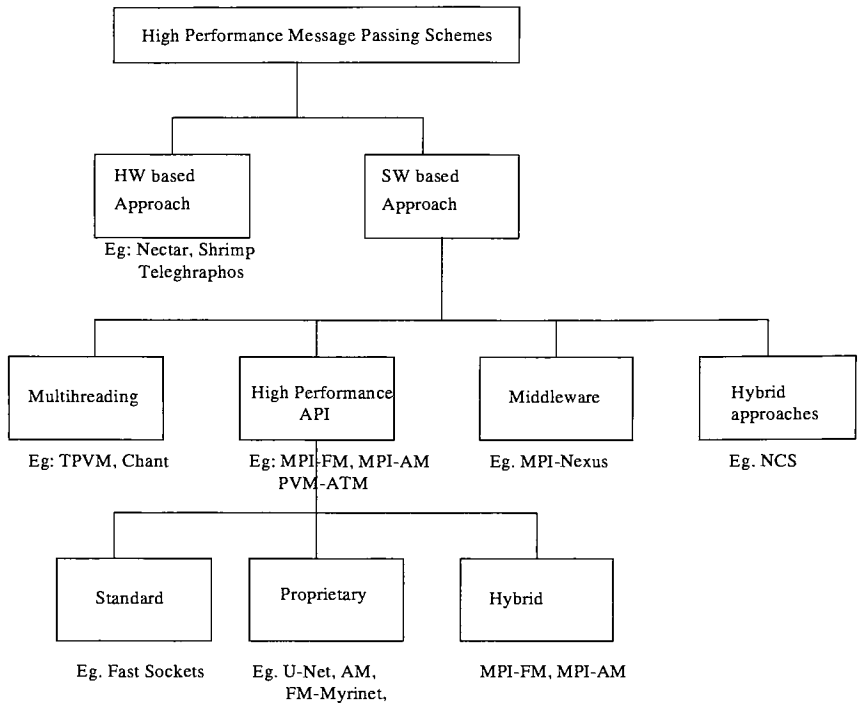


Figure 13: *Classification of High Performance Message-Passing Schemes*

2.4.1 Hardware Based Solutions

Hardware based approaches focus on building special hardware [9] to reduce the communication latency and achieve high throughput. The developers of communication hardware should implement the device drivers and proprietary APIs for their communication hardware. However the constraint to use special communication hardware makes it difficult to port existing applications to a different computing platform.

2.4.2 Software Based Solutions

On the other hand software based approaches incorporate one or more of the following software techniques.

Optimized implementations of kernel protocols.

Other low level kernel messaging layers.

An efficient user level protocol layering

High performance APIs.

Multithreading and utilizing middle-ware services to utilize existing network interfaces.

The most common and successful technique is to replace standard communication interfaces (Eg. Socket Interface) used in existing message passing systems with high performance communication APIs. (Fast Sockets[34], FM[8], U-Net,[33] Active Messages[27] and others).

Fast Sockets[34]:

Fast Sockets, based on Active messages, is an implementation of the Berkeley Sockets on top of its lightweight protocol. The Fast Sockets user space library explores "an advance receive buffer posting" copy avoidance technique to eliminate the extra copy at the receive site if possible. (Figure 14) When the packet arrives, for any reason if the receive buffer is not yet posted, or it is not possible to DMA to receiver buffers addresses, the message handler copies it to the user buffer.(Figure 15)

Figure 14: *Data transfer via receive posting (True zero copy data transfer)*

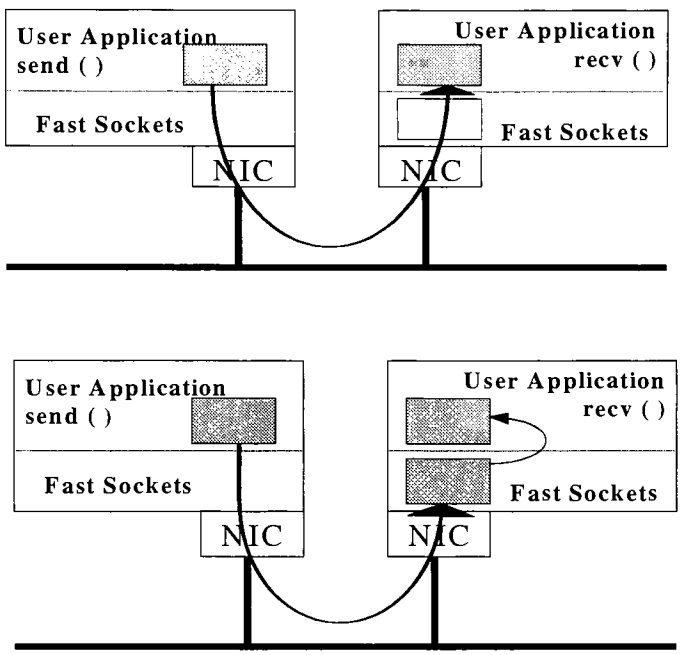


Figure 15: *Data transfer in Fast Sockets.*

FM[8]:

Illinois Fast Messages (FM) provides a streamlined interface similar to that of Fast Socket. The main difference is that Fast Messages supports packetization and thus works with messages of arbitrary message size. MPI ADI is also created for Fast Messages 2.0 and uses the streams abstraction to offer layer interleaving of FM's and the application thread of execution on the receiver site. A typical message processing scenario within the receiver packet handler is illustrated below: [8]

```
int  myHandler(FM_stream, *str, unsigned sender)
{
      struct header   myheader;
      int  msglen;
      /* Get the header      */
      FM_receive(&myheader, str,
```

```
                              sizeof(struct header));
        msglen = myHeader.length;
        if(myheader.littlemsg)
              /* Short Messages  */
              FM_receive(littlebuf++,str,msglen)
        else  /* Long Messages */
              FM_receive(findbuf(msglen),str,msglen)

        return FM_CONTINUE;
}
```

The first FM_Receive() call is used to extract the message header. Then the handler reads the header fields, identifies the message, and selects the buffer into which to copy the message payload. The stream implementations of MPI-GB use a similar approach with real time scheduling of the message handler. Note that FM requires one copy from the streams buffer to the posted receive buffer if not running on top of the Myrinet. FM uses scatter-gather approach to fragment and reassemble the packets larger than 1Kbyte.

In MPI-GB, this threshold is set to the maximum transmit size of FDDI MTU, 4352 bytes. Like Fast Sockets, FM does not offer any features for real-time scheduling of handler and therefore providing support for guaranteed delivery. Fast Sockets and Fast Messages provide receive flow control, allowing the receiver to control the rate at which data is processed from the network. This approach eliminates network overruns of application buffer pools, thereby avoiding memory copies. MPI-GB uses the same approach by checking the buffer low and high water marks and sending a negative acknowledgement to the sender if the high water mark is reached.

U-Net[33]:

U-Net, developed originally for ATM networks, is another high performance messaging protocol and API. It provides a true zero copy, low latency communication as well as

flexible access to the lowest layer of network layer. Like the memory map version of MPI-GB and contrary to other versions of MPI-GB and FM, U-Net tries to avoid the passage of data through kernel memory by performing DMA transfers directly into the user buffers. The problem with this approach is that, the user must allocate a physically contiguous buffer and pin it down to a physical memory address. In MPI the buffers needed by applications and their locations are generally not known in advance. There is an on-going research effort for U-Net to dynamically map and unmap those user buffers using TLB. Thus messages can be transferred to/from a memory allocated after the initialization phase in the MPI application.

Because of the time constraints, MPI-GB will not include this feature in the current release. MPI-GB assumes that all of the receive and transmit buffers are defined before the initialization phase and so that they can be mapped to the board's DMA engine in the MPI process startup

NCS[12]:

The implementations discussed so far do not address the issues of quality of service (QoS). In order to provide the QoS to distributed applications, issues such as scheduling, context switching and synchronous bandwidth allocation should be addressed. Syracuse University's NCS implementation [12] tries to eliminate the overhead of OS scheduling and context switching by send and receive trap routines to invoke the device driver's read and write routines directly.

This requires changing the trap vector table in using the kernel debugger (ADB). This approach creates an overhead at the receive site since it only allows it to eliminate the

overhead after an inquiry to receive the packet is made. This received packet may be waiting on kernel buffers for some time. MPI-GB approaches this problem by raising the priority to maximum for the sleeping process that is waiting for this packet in the device driver's interrupt service routine. Therefore the blocked receive process can be awakened immediately.

fbufs[31]:

The other approach to fast communication is to tune existing kernel-mode protocol implementations like tcp/udp to deliver more performance. The implementations using this approach, try to eliminate the copying of the data presented by the user applications for output, and for data obtained from network devices and protocols on input operations. New Solaris OS 2.6 release contains zero copy tcp[32] which uses a similar approach to fbufs. This TCP implementation re-maps pages of data from one domain to another instead of copying. It uses copy-on-write pages to prevent the user application from corrupting data still live in the protocol stack.

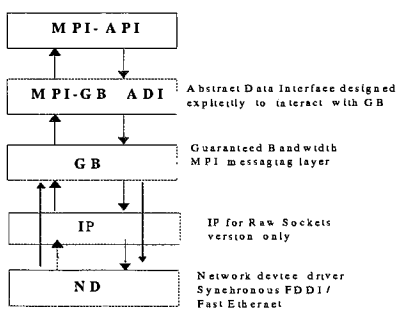
3. MPI-GB Design

3.1 MPI-GB Design Methodology

After carefully investigating the problem areas (Chapter 1, 2), MPI-GB is designed and implemented step-by-step using divide and conquer approach.

The overall architecture of MPI-GB is given in Figure 16. MPI-GB ADI, GB protocol, network device driver and interfaces among these components are designed in this project as a part of this thesis.

Figure 16: *Top Level MPI-GB Design Blocks*



In this chapter, the bits and pieces of MPI-GB architecture will be explained. MPI-GB interface to MPI API, GB components, GB device driver interface are presented. Details will be given regarding the incremental approach taken during development of three different versions of MPI-GB resulting from modifications.

3.2 MPI-GB Packet Structures: MPI, GB Packet Header. IEEE 802.3 Encapsulation

3.2.1 Notational Conventions

This section utilizes specific notation, terminology, and illustrations to produce a consistent protocol description as described by Internet Engineering Task Force (IETF). The conventions are described below:

- Terminology

  The following terms refer to data objects within the packet: **word** indicates a 4-byte (32 bit) object, **field** refers to object any size, **bitfield** denotes a field contained within a word, and a **segment** denotes an object consisting of one or more fields.

- Unless specified otherwise, packet structures are discussed and portrayed with the most significant and/or byte on the left (MSB)

- Illustrations

  Within illustrations, objects drawn as ellipses contain additional structures; objects drawn as rectangles contain no substructures

- Notation for bitfields
  - Bits are referred to by name and are annotated in capital letters of a distinguished font style e.g. **NAME.**
  - Within illustrations, vertical lettering (representing the label/function) beneath a numeral indicates a single-bit field.
  - The number of bits in a bitfield, when designated, is with colon syntax: e.g., name:7

- Within the text, multiple-bit fields are labeled in lower-case, italicized characters: e.g., *name*.
- Within the text, the meaning of certain fields is determined the setting of a designated bit within the field. An apostrophe convention is used with these fields to indicate whether the bit is set. The field with the bit set is shown as "primed" by placing an apostrophe (') after the field: e.g., K is an unprimed field and K' (K-prime) is the field when the most significant bit is set.

- Notation for bytes:

  - Objects not specifically designated as single-bit fields represent sections of a packet made up more than one bit or of "n" bytes.
  - The number of bytes in a field is indicated by a numeral enclosed within parentheses: e.g., *name*(4) is 4 byte field.

    The letter "n" is used to indicate the size of bytes of a variable-length items: e.g., *name*(n). The form (8*n) designates a length that is multiple of 8 bytes.

3.2.2 MPI-GB Packet and FDDI Encapsulation

An MPI-GB packet is contained, or encapsulated, within a Medium Access Control (MAC) layer frame as shown in Figure 17*. MPI-GB can be designed to be independent of physical layer influences, but it does depend on the device layer addresse, Frame Control (FC) field and MAC address section of the FDDI header.

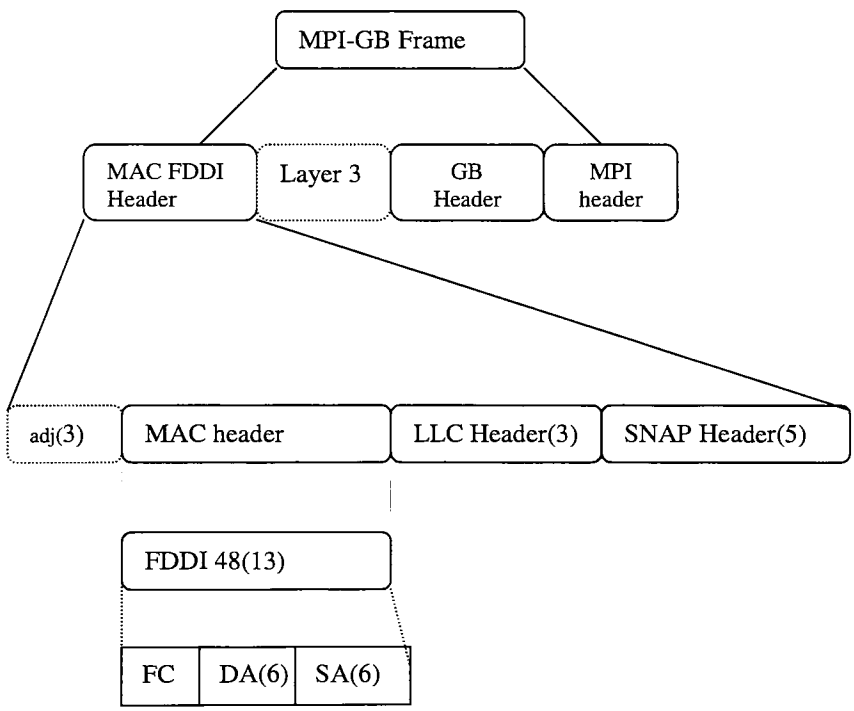
* In this thesis, FDDI encapsulation will be used to replace the MAC header. Nevertheless, any physical MAC layer encapsulation such as Ethernet, LLC, IEEE 802.3 can be used as a replacement.

The "layer 3" box in Figure 17 is drawn with the dashed line. As explained earlier, the first version of the MPI-GB implementation uses raw IP sockets to talk to the device driver, the second and third versions talk to the device drivers directly bypassing layer 3.

MPI-GB capsulation of FDDI follows the guidelines established in RFC 1103 " A proposed Standard for the Transmission of IP datagrams over FDDI networks." This encapsulation frame includes the LLC and SNAP headers as defined in Figure 18.

The device driver applies a 3-byte offset to the start of the FDDI header to align protocol (IP/GB) packet.

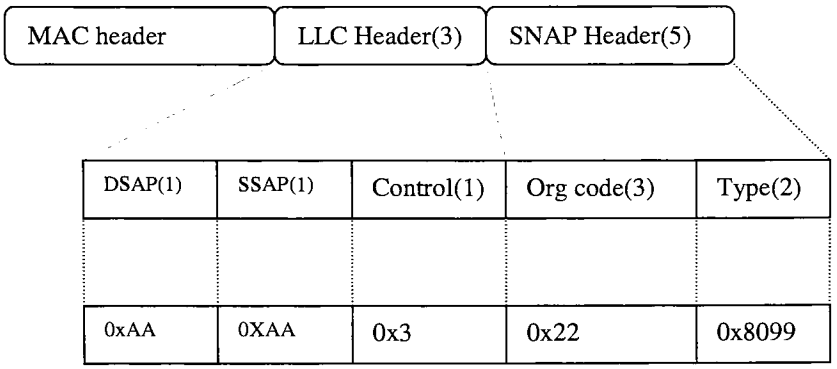
Figure 17: *MPI-GB, MAC FDDI Header*



FC: Ex: 57 for Async Traffic
D7 for Sync Traffic

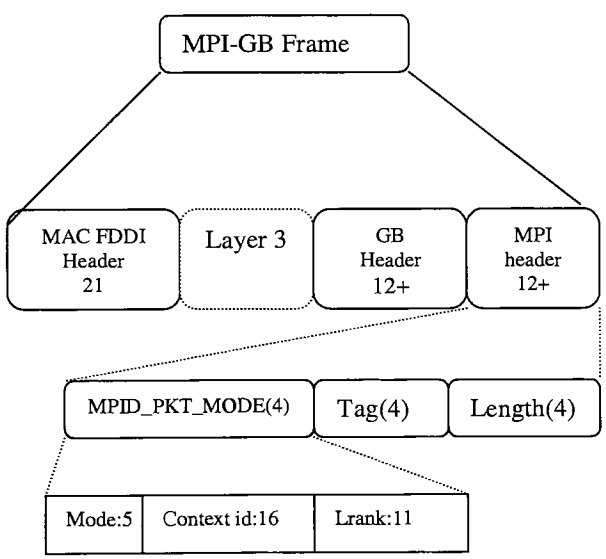
DA Destination Address
SA: Source Address

Figure 18: *LLC and SNAP header used in MPI-GB*




```
OrgCode:          Unique number 0x22 chosen for GB protocol.
Type:             0x8099 : Chosen for GB packet.
                  0x8060 : For ARP packet.
                  0x8000 : For IP packet.
```

***Figure 19:** MPI header*



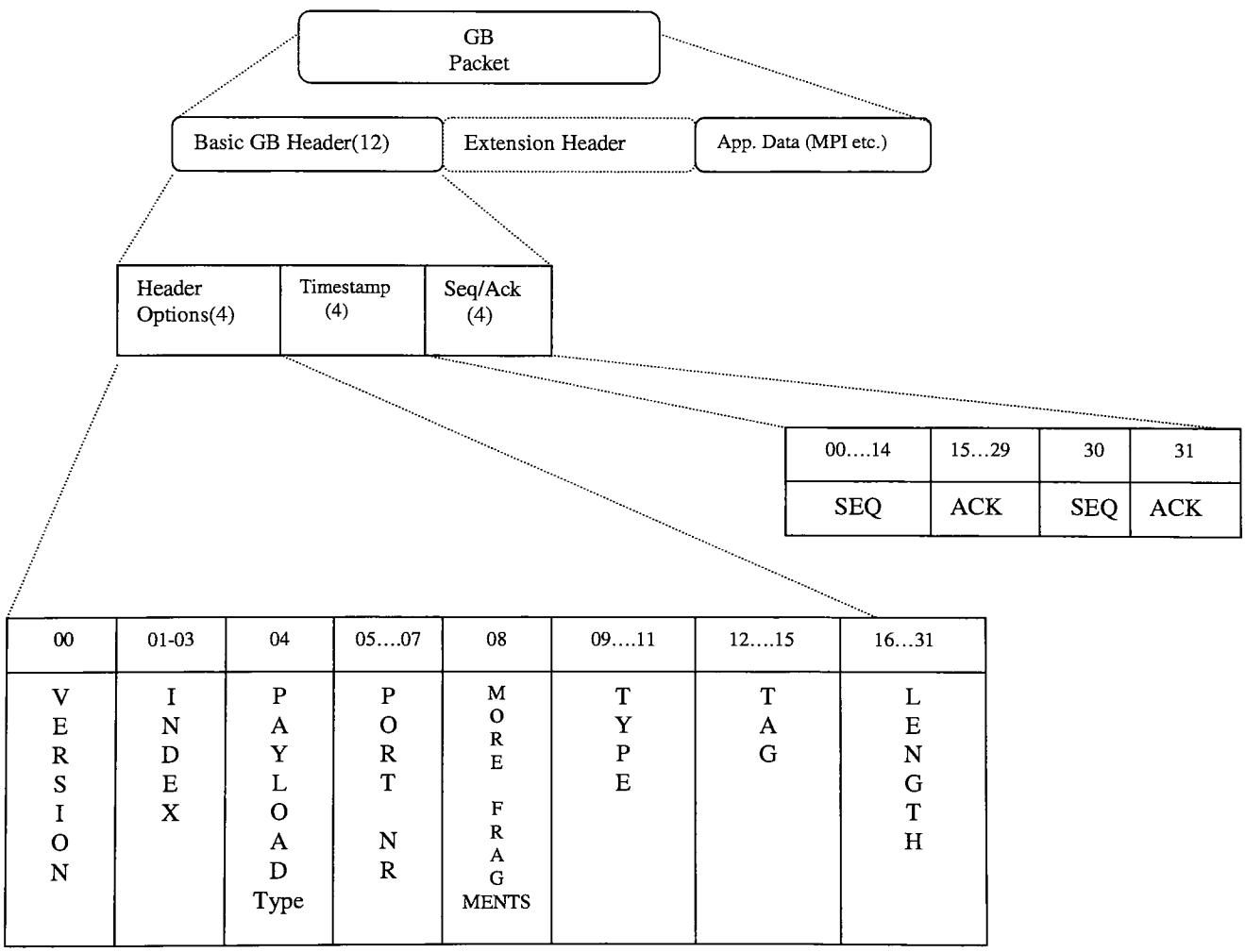
An MPI message consists of data, the length field, a message tag used to distinguish between messages, context-id, mode and a destination for sending or source for receiving.

***Table 3:** MPI header fields*

MPI Header Fields	Meaning
Tag	Tag value (Like type) used to distinguish messages.
Context_id	Context id
Mode	Sending mode (Standard, Ready, Sync, Buffered)
Length	Length of the packet.
Lrank	Local Rank in Sending Context (Destination/Source Rank)

Messages can be received in the API only exactly matching the context-id exactly matches, and either the tag and source matches or specifying that any source and/or tag may be matched.

Figure 20: *GB Header*



Seq/Ack field caries 15 bit Sequence and acknowledge numbers along with SEQ and ACK bit indicating the validity of these fields.

Version: shows the version of GB implementation.

Index: indicates a translation table index for the handler table.

Payload Type: indicates the message is synchronous or asynchronous.

Port Number: It is intended to be used to differentiate between multiple processes using GB. The current version of GB allows only one process running on a host. Port Number is currently assigned to 7, which means "not applicable"
Subject to change with upcoming revisions

More Fragments: Used for fragmentation and reassembly.

Type: Shows whether the packet is a data or control packet.

Tag: Same meaning with MPI, used to distinguish messages.

Length: Length of the packet including GB and MPI headers and excluding FDDI header

Timestamp: Each packet is marked with a timestamp before sending it to the network, it is used for guaranteed delivery insurance and testing.

3.3 MPI-GB Protocol Description

3.3.1 Initialization

Although it may look very simple to application developers, the initialization process is one of the most complex processes of MPI-GB. MPI_GB_Init and MPI_GB_Commit calls are provided to applications to perform these tasks.

MPI-GB_INIT

The application calls the MPI_GB_Init ADI macro to start the ADI, GB processes. This macro subsequently calls GB_Init. This Function is responsible for starting up the GB processes, initializing the MPI-GB, starting the SBA state machine if it is not already started and setting up a stream/mmap interface to driver.

The following information is required to create a MPI-GB connection between all the nodes:

- The local and remote host's communication port numbers
- The remote host's MAC address

IP addresses of remote hosts are given with a common table. GB_Init performs the following tasks:

- The Originator (master) queries its MAC address from DLPI and opens a stream channel to device driver for each slave in the hosts file.
- The master reads the names of the slaves from the host file, and starts up the slaves with rsh, passing its hostname as an argument.
- When the slaves come up, they also learn about their MAC address and open the driver with a DLPI calls.
- The slaves set up a tcp connection to the master and notify MAC addresses and port numbers.
- The master sends out all the necessary information to the slaves using a raw connection. At this time, all the necessary information including the MAC addresses and port numbers are known both by the slaves and the master.

Please note that TCP/IP is used for setting up the streams and distributing a-priori knowledge. Knowing that the start-up phase is not performance critical, the reliability and ease of usage are the reasons for TCP/IP selection startup phase.

Once again, the current implementations of GB does not allow multiple processes to be running at the same time. Nevertheless port number notification is provided anyhow, in case the protocol is updated to support this feature in the next revisions.

MPI-GB_COMMIT(GB_resource_struct *)

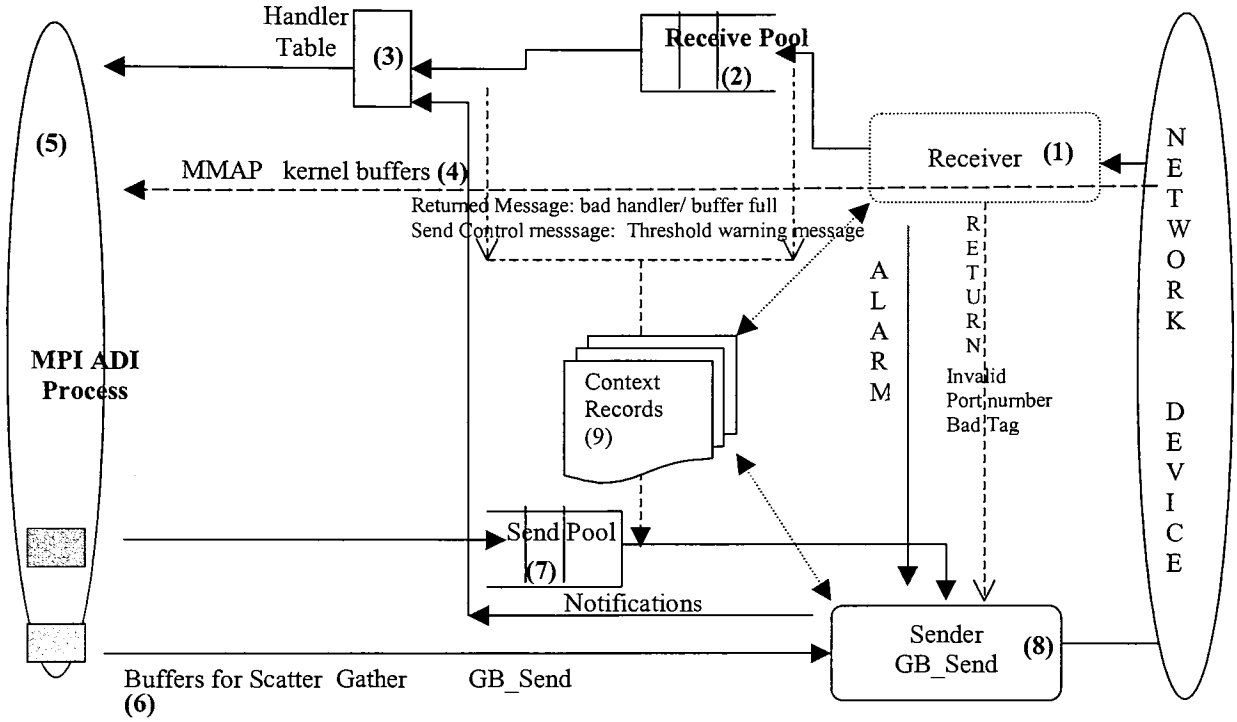
```
typedef  struct {
                char      *send_buffers;
                int       slength;
                char      *rec_buffers;
                int       rlength;
                int       pay_type;
                int       payload;
            } GB_resource_struct;
```

The application calls this function with a set of resources as an argument. GB_Commit is called from MPI-GB API with the given resources as arguments – e.g. payload type, buffers, length of the buffers, and payload unit. The application uses this function to register system resources such as network bandwidth and memory in a controlled and predictable manner so that the communication operations complete within the QoS requirements.

Buffer and bandwidth reservation with this command will be explained in the following sections.

3.3.2 GB Processes and Components

Figure 21: Inward Architecture of GB



Component (1): Receiver

The receiver acts on packets received from the network. It identifies the message and calls the appropriate handler based on the message index value. It stores the buffer pointer and control state for a received message to a receiver buffer pool. Receiver functionality is integrated into the device driver in the MPI-GB MMAP implementation to minimize context switching costs. In this case, the device driver has access to the handler table and triggers the handler based on a message tag. The receive buffer pool is shared by the handler and the device driver. GB uses similar approach to active messages[27] for a receive packet handling.

If the receiver identifies a bad tag or bad port number, it returns the message. It is also responsible for assembly of the data, error, flow and rate control. If it detects an alarm condition (such as the maximum threshold is reached in the receive buffer queue) or an error (such as a missing sequence number in the packet), it alarms the sender to take appropriate action.

Component (2): Receiver Pool Queue

This pool is separate from the receive post queue used in MPI. It contains a pointer to the control buffer structure, which includes a pointer to the receive data buffer and the control state of the received buffer. This control state is updated when MPI API consumes the message. The receive pool queue is maintained internally.

Component (3): Handler Table

The handler table is responsible for translating handler indices to handler functions. The handler table is filled at *init* time. Messages carry an index into handler tables.

For example for short /long / control messages different handlers are used.

Component (4): Receive Memory

Received data is copied from Streams kernel memory for MPI-GB stream implementations. For the third version, MMAP implementation, the device directly Scatter-Gather DMAs to the receive memory.

Component (5): MPI –GB Abstract Device Interface

The GB directly interfaces with MPI-GB Abstract Device Interface. The MPI-GB Abstract device interface functions as a bridge between MPI API and GB. The following section describes this interface. The Appendix includes function definitions of MPI-GB ADI.

Component (6): Transmit Memory

The transmit user buffer is copied to Streams kernel memory for MPI-GB streams implementations. For the third version, MMAP implementation, the device directly Scatter-Gather DMAs from the transmit memory.

Component (7): Send Pool

This pool is separate from the receive post queue used in MPI. It contains a pointer to the buffer structure, which includes a pointer to send buffer and control state of the sender buffer. This control state value can take three values:

0:The data is sent but an interrupt from the board has not been received yet.

1:The interrupt is received. The data in the buffer is still valid, not overwritten

2:The data in the buffer is invalid, it was overwritten by the sender.

Component (8): Sender

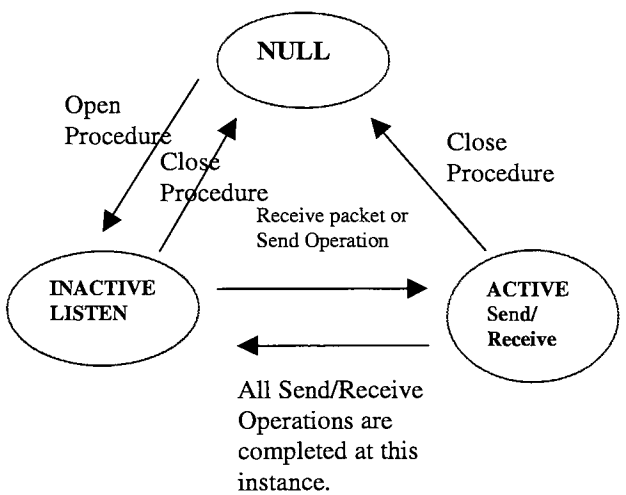
Sender formats the data and control packets. It prepares the GB header for each packet and passes the MPI, GB, FDDI headers and data to network protocol interface. The network protocol interface is the Sockets API for IP Sockets implementation, and

Streams Data Link Provider Interface(DLPI) for the DLPI implementation. For MMAP, GB_Send directly traps into the kernel. The device DMAs directly from the send and header buffers utilizing Scatter-Gather functionality.

Component (9): Context Records

Context records are maintained for each connection. Context may be in one of the following states: NULL, Active (Send/Receive), Inactive Listen state. All contexts are initially in the NULL State. A Send operation or received packet that invokes the receiver on an Inactive Listen context takes it to the Active state. If GB is not actively sending or receiving any data, it is in the Inactive Listen State. GB may be sending or receiving data during the active state.

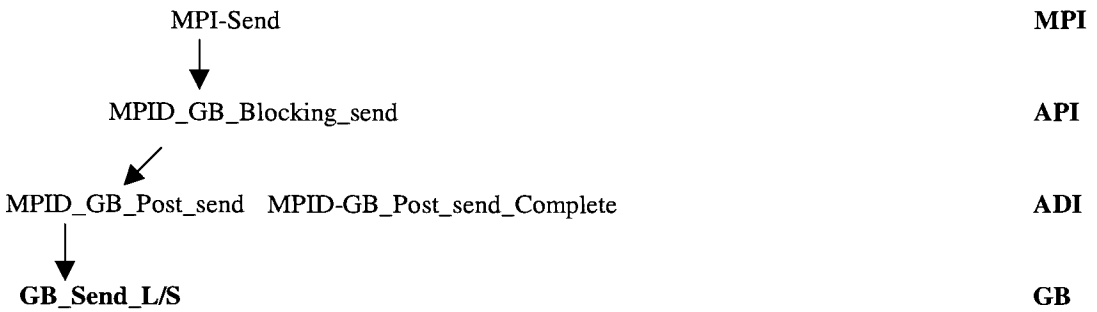
Figure 22: *GB States*



3.3.3 MPI-GB ADI

GB is interfaced with MPI in an efficient way to reduce latencies by taking advantage of FDDI and GB's capabilities. The interface between MPI and GB is designed to provide mapping between MPI and GB primitives.

Figure 23: MPI to GB Send Flow



As shown in figure 23, there is no one-to-one mapping between GB and MPI routines. GB does not have split phase operation like ADI (post_send and complete_send) It has one send operation that starts and completes the send. The split-phase nature of the ADI communication primitives implies that a state must be maintained for each pending operation. For this purpose, to store the state of each request, a separate queue has been employed on the sender and receiver site.

In MPI, a message may be sent in either blocking (MPI-send) or non-blocking (MPI-Isend) form. In the blocking form (Figure 23), the ADI will not return control to the API until the message body is available for re-use. The MPI specification gives freedom to the ADI designer to choose to wait either until a message buffer has been delivered or has been copied out of the MPI memory. The second method has been chosen in MPI-GB ADI implementation.

The API requests the ADI to send a message by forming a **Request** structure containing the information in Table 4 as described in the ADI spec [20]. API will initialize the ADI data area in the structure (the dev_shandle). Then NonBlocking send is called with flag=1

if the send is nonblocking and flag=0 otherwise. *Post_send* is called subsequently with the request structure argument. The *GB_send* is called subsequently to send the message. *Post_send* returns to the caller immediately after *setting* the send queue's pending flag for this request. If the request is blocking, API calls *Complete_Send*. The ADI does not return from this call until the send has "completed". Note that the "completed" flag in MPI-GB means that the device already completed the DMA of the data (MMAP version) or copied the message to the streams queue (Streams implementations). In both cases this means to ADI does not to return until the message data buffer is available for re-use. The nonblocking send operation design table given in ADI spec[20], is updated for MPI-GB implementation (Table 5).

Actually GB has two send routines, one for short and another one for long packets, which exceed the FDDI MTU (4352) size. (This will be explained in more detail in the following sections.)

Table 4 : *Request information fields*

Request Field	Meaning
Handle_type	Type of handle (MPIR_Send or MPIR_RECV)
Dest	Destination Rank (send)
Source	Source Rank (receive)
Tag	Tag value
Context_id	Context id
Completed	Flag for whether communication operation completed
Mode	Sending mode (Standard, Ready, Sync)
Dev_shandle	Device's send handle
Dev_rhandle	Device's receive handle
Datatype	MPI-style datatype description

The MPI-GB implementation tries to avoid redundant copying of the buffer in the send site. MPI API adds its own header to a GB message before sending the packet to another

node. This header contains information about the message being transmitted, so that MPI running on the receive site can identify the message. Due the nature of the MPI implementation, the header and the data reside in different locations.

The intermediate buffer is needed to assemble this header with the data before sending it to the network. As for the receive side, the packet needs to be disassembled before processing the packet. (Receive side operations will be explained later in this chapter).

To avoid the extra copying on the send side, the *GB_send* operation is designed to take two buffer pointers (MPI header and Payload). Later these scattered buffers are gathered in the FDDI device and sent as one packet (DMA as one piece). We believe that delaying gathering of these pieces to the network hardware (Scatter-Gather DMA method) helped us to avoid a major load/performance drop-back. By utilizing the Scatter-Gather capability of the hardware, we were able to eliminate the extra load that might have been caused by copying these pieces into the temporary buffer.

Table 5: *Non-Blocking Send with MPI-GB*

User MPI Program (Non Blocking send)	MPI Implementation (MPICH API)	ADI device MPI-GB
MPI-Isend	*A alloc*_send allocated *send*_handle *posts* _send calls device layer to start the send operation	MPID_GB_Post_send Initiates send operation after getting the necessary data Hands into the GB, then DMA'd to the device
(User Code Runs)	Return	
	Posts send completed in MPI data structures. Frees device handle structures	Interrupt from the board indicating that the message is sent; calls mark_send_completed..
MPI_status	MPI data structures show send completed	
MPI_wait	D_SHANDLE found marked completed	

Receiving a message is much like sending one. The progress of unblocking receive is shown in Table 6. A comparison send operation given with Table 6 shows that the only significant difference is the "check unexpected queue;" this handles the case of the data having arrived before the user posts the receive for the message.

Instead of an explicit receive operation GB relies on user-defined "handlers" to handle the content of the received messages. These handlers are installed during the MPI-GB initialization with GB_SetHandler calls. MPI-GB uses separate handlers for short and long packets as long packets require assembly. GB_receive_handler(s) are not directly called from the ADI routines, therefore it is connected with a dashed line to ADI in Figure 24. This handler's responsibility is to copy the content of the incoming message in the destination buffer whose pointer is found in the receive request from API for that specific message. In case this pointer is not found, the message is stored in the temporary buffer and its availability is marked in the unexpected queue. API first checks the unexpected queue before posting a receive request. The operations of anonblocking receive in case the message has not arrived is shown in Table 7.

Figure 24: *MPI to GB Receive Flow for blocking Receive*

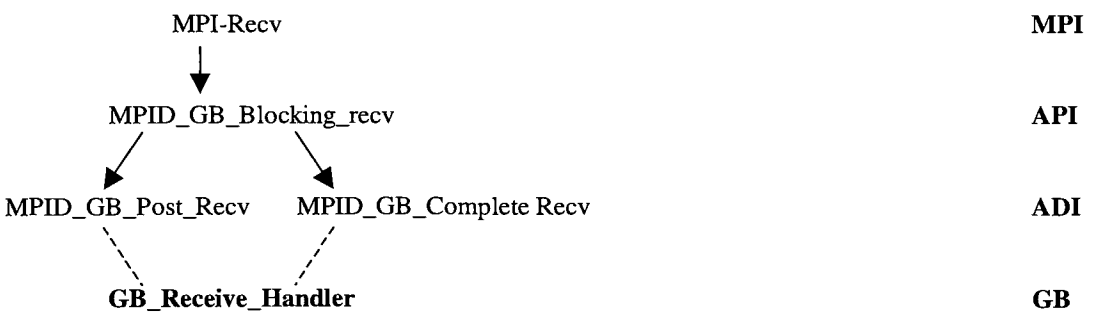


Table 6: *Non-Blocking Receive with MPI-GB*

User MPI Program	MPI Implementation (MPICH)	ADI device MPI-GB
MPIirecv		
	Check unexpected queue (suppose not found) *Alloc*_recv_handle Set fields, particulary "non-blocking"	
	return …	
User code		_GB_post_receive Posts receives in the receive queue at the device level
		…..
		(message arrives, interrupt calls from device) Handler looks at the header and posts to the appropriate buffer
	Marks receive complete	
MPI_status		
	Check_device check MPI data structures for status return	
MPI_wait		
	(waiting on a particular receive transfer control to device layer using complete_receive (poll) return	

The mirror of the problem faced on the send side is tackled for the receive side. Long packets (>MTU) are broken into packets on the send side. They need to be reassembled on the receive side. A simple solution is to utilize a temporary buffer to put those buffers back together on the receive side. This requires an additional copy for long packets before the application handler is called with respect to short messages.

The goal is to reassemble the message directly into the application message buffer. But because of the distinct layering mechanism of MPI, the destination buffer is only known to application (MPI) and the changes are necessary to allow GB to copy the payload to this buffer.

GB uses a similar approach to the FM [8] to tackle this problem. A generic handler is written to extract the messages. After receiving, analyzing and identifying the message header only, it selects the buffer into which to copy the message payload. Finally the second receive is executed with the selected buffer as an argument in order to extract the payload directly into the buffer. This buffer is obtained by looking into the posted receive queue. If the receive is already posted by the API, the payload is received directly to application buffer. If not, it is temporarily posted to the unexpected queue. Please note that the handler is called directly as soon as the first packet is received and the DMA is started. Since packets belonging to different messages can be received interleaved, the execution of several handlers can be pending simultaneously. Also on a long message the handler can be processing one part of the message while the sender is

transmitting the rest. This level of multithreading brings a number of performance benefits to MPI-GB architecture.

3.3.4 Buffers, Packets and Messages and Message Queues

Dynamic memory allocation is a major source of unpredictability during application execution. For example, the memory required for a critical communication operation may not be available. Such unpredictability makes it difficult for a system to satisfy an application's QoS requirements. MPI-GB utilizes the concept of static buffer and buffer management for control and payload messages.

The application should register all payload buffers by the *GB_commit* command. Payload buffers are represented by (address, length) pair with the understanding that it implies a set of pages and/or scatter gather list. Once the descriptors are filled, and DMA is started, the device performs the scatter-gather DMA operation to/from those buffers. In addition to payload buffers, buffers that include header and other control information are statically allocated and mapped into devices at MPI_INIT time. Special care is given in the design process to avoid any redundant data copying in both MPI and in the device interface.

As discussed in a previous section, in MPI implementation, there are three control message request queues to store the state of each pending request:

One on the send side for posted sends and two on the receive side: one for pending receives (posted receives) and one for unexpected messages. (Ones that have been delivered, at least in part, but for which the API has not yet issued a matching receive.)

The "Unexpected queue" is needed to store unexpected messages, i.e incoming messages for which a receive request has not yet been posted. This queue would not be needed if messages were buffered on the send side and then transferred upon posting the receive request. The unexpected queue method is chosen to avoid extra latency that would result in two-way communication. (Table 7)

Table 7: *Non-Blocking Receive is posted after the message arrives to MPI-GB*

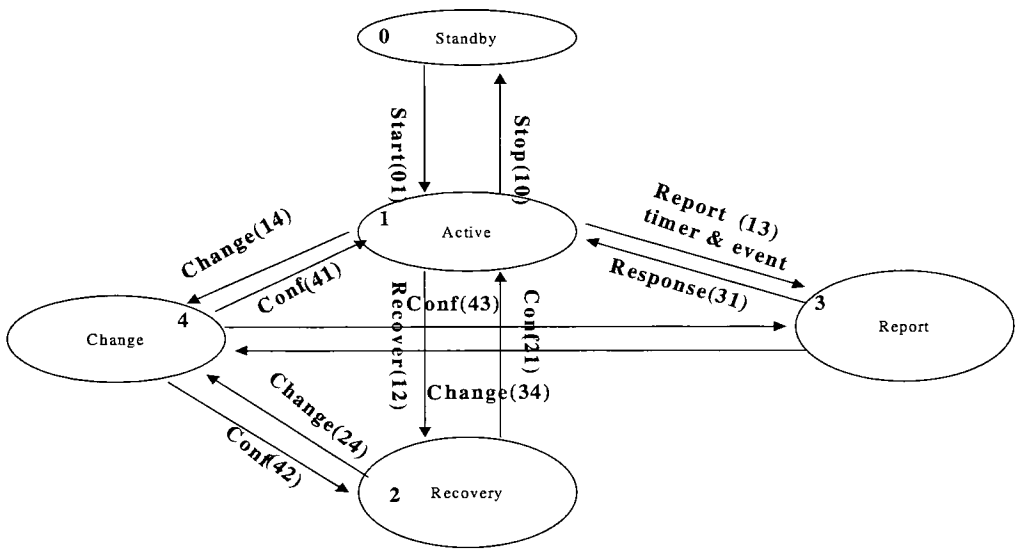
User MPI Program	MPI Implementation (MPICH)	ADI device MPI-GB
		(Message arrives) Handler calls D_msg_arrived.
	A_alloc_recv_handle in Unexpected queue	
		D_msg_arrived returns status of "not posted". This searches first the posted receive queue, and, if not found, The device driver transfers into the unexpected queue. D_mark_receive_completed
MPIirecv	Checks the unexpected Queue, finds the message A_complete_receive	
MPI_status		
	Return "completed"	

3.3.5 Synchronous Bandwidth Allocation and Bandwidth Reservation

The FDDI MAC essentially supports two classes of traffic: synchronous and asynchronous. The synchronous class of traffic is essentially guaranteed a pool of bandwidth at all times. The asynchronous class of traffic gets the remaining bandwidth. FDDI device driver utilizes the SMT 7.3 implementation. MPI-GB configures the SBA state machines for the dynamic allocation using the MPI_GB_COMMIT command. SBA payload can be configured up to 50 Mbit/sec, which equals to half of the FDDI bandwidth.

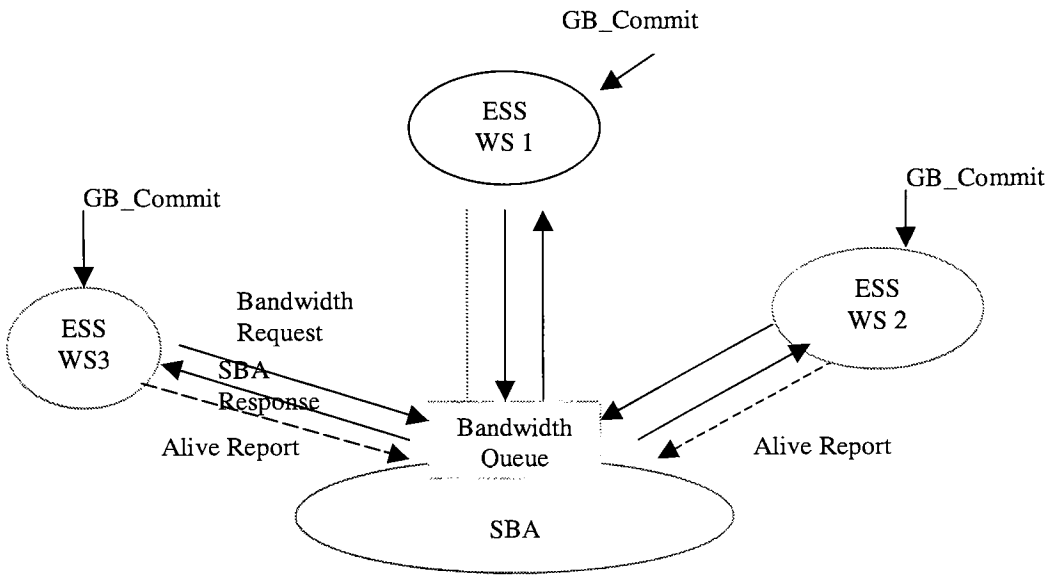
Synchronous Bandwidth Allocation protocol utilizes two state machines: Synchronous Bandwidth Allocator (SBA) and End Station Support (ESS). SBA acts as a bandwidth server and allocates bandwidth to nodes in FDDI ring as requested by the ESS. In MPI-GB implementation, SBA is started upon sending MPI_INIT command. Generic SBA state machine diagram is shown in Figure 25.

Figure 25: SBA state machine diagram



As soon as ESS state machine receives the configuration requests from the application, it forms a bandwidth request packet and sends it to the station where the SBA is running. SBA, upon receiving this packet, analyzes the request. SBA can approve, disapprove or reject this request depending on the available bandwidth. ESS Clients periodically send packets to SBA indicating that they are still alive and using the bandwidth. If the SBA does not receive any indicator packet from a particular ESS client for some time, it assumes that this bandwidth is available for new ESS requests.

Figure 26: *SBA ESS Relation*



The application calls this function with a set of resources as an argument. Then, GB_Commit is called with the payload information. This information is used to program the SBA State machine's *sbaPayload* attribute. This attribute defines the requested synchronous bandwidth for manual static allocations in SU (Synchronous Units). The Synchronous Unit is the number of bytes transmitted in 125 microseconds. The correlation between payload given in Mbit/sec and in Synchronous Units as specified in ANSI Standard is shown in the following table.

Table 8: *Payload Synchronous Unit Correlation*

Mbits/s	1	2	3	4	5	10	15	20	25	30	35	40	45	50
Payload	16	32	47	63	79	157	235	313	391	469	547	625	704	782

3.3.6 Message Prioritization and Scheduling for Synch/Asynchronous Traffic

Message prioritization is fully supported in FDDI networks. The FDDI hardware assigns a high priority to synchronous packets and ensures that all such packets are sent before asynchronous packets can be transmitted. As described in the previous chapter, SBA and ESS state machines are provided to configure and manage the synchronous bandwidth allocation in the ring. FDDI hardware also allows for prioritization among the asynchronous packets.

MPI-GB design is motivated by the need to control latencies and delays for distributed real-time applications. This leads to a requirement bringing the hardware message prioritization and scheduling abilities to the user application. During the device driver and protocol development of all three versions, the following precautions are taken for all three versions of MPI-GB during the device driver and protocol development to serve this goal:

- Synchronous and asynchronous buffer queue and transmission paths are separated from each other. Therefore synchronous packet processing is not affected by asynchronous packet handling.

- The first two versions rely on the Solaris streams scheduler to schedule device driver's ***wput*** routine from protocol's ***putmsg*** call in its send routine; Likewise, on the read site (upstream), the driver receives the data via device interrupts. The read-site ***rput*** procedures run at interrupt level, and hence can not afford to block. Driver's **rsrv** routine is scheduled by Solaris scheduler in the system context.

Streams scheduling is implemented a routine called ***runqueues***( ) which has no relation to UNIX process scheduling. The Solaris OS™ streams scheduler is not designed for real time scheduling. As documented in Solaris manuals [22] the quanta vary from 20 ms to 200 ms; this of course can increase latencies in the MPI-GB protocol and nullify all the good work that has been done. To avoid scheduling, context switching and redundant copy overhead caused by streams, the shared memory (MMAP) version of the driver was implemented. This driver can also be named as Character device driver. Stream framework does not allow an MMAP interface.

In this model, as a first attempt to avoid the context switching and scheduling latency between user and kernel space, send trap and trap handler routines are implemented. This allowed the send process to switch to kernel mode immediately. As we do not have access to kernel code, this is done via modifying the trap table and adding trap handler code to the kernel using the kernel debugger (KADB). Although we were able to send data using this model, we faced mysterious page faults and crashes from time to time. As we do not have access to the Solaris kernel code, this problem can not be resolved.

Solaris as a UNIX System V Release 4 implementation uses a notion of priority classes and supports three types: Real-time, System and Time-sharing. When a process is created, it inherits its parent's priority class scheduling, which includes its priority class and priority value within the class. The process remains in that class unless it is changed as the result of user-mode request using the ***priocntl*** call. As a second attempt, the GB sets itself to a minimum allowed real-time priority using ***sched_setschedule*** call. This

assigns the minimum ***rt_quantum*** value, which is the maximum number of clock ticks that can elapse while this process uses the CPU. While there is a process on the dispatch queue that is in the real time class, no other system or time-shared class process is scheduled.

The average Roundtrip latency and the percentage of messages, which miss their deadlines are not affected much with the asynchronous performance load after these modifications. (Figure 34 and 35).

3.3.7 Flow /Rate and Error Detection

The volume of GB output is regulated by sequence numbering, threshold mechanism and negative acknowledgments provided by the receiver. A sequence number is defined for each output packet, starting with the initialized sequence values.

The GB provides a receiver flow control, allowing the receiver to control the rate at which data will is processed from the network. This feature is only possible because of the underlying reliable delivery provided by FDDI.

If the predetermined number of buffers currently in use exceeds the threshold value on the receive side, the receive side sends a control packet to the sender requesting it to slow down. Depending on the pre-selection, the sender may decide to decrease the bandwidth or terminate the data transfer. If the sender ignores the warning and continues sending the packets with the same speed, the receiver will send back the packet if it does not have any space. So only the receiver rejected packets have to be re-queued at the transmit site. It is

very unlikely, but if it happens, GB leaves the packet re-ordering task to the application (MPI API) as the packets arrive. Although MPI-GB checks the sequence numbers to ensure that all the packets are received in order, it does not attempt to reorder the packets. Depending on the pre-selection it may instead send an error message to the application and fail, or it may leave the packet reordering task to the MPI API as explained above.

If the receiver detects a missing sequence number and has not sent any warning control messages lately, a negative acknowledgment is sent to the transmit end. Depending on the window size, if this packet is still available (not overwritten), the transmission is repeated from this point. If the buffer was overwritten, the connection closes and an error message is sent to the application.

In summary, GB does not provide a full error control mechanism for lost or damaged packets due to network problems. It depends on the reliability provided by FDDI. This is one of the fields where GB can be improved in future releases. On the other hand GB fully supports error recovery due to the buffer shortage problems that may occur on the transmit or receive end.

The flow control explained in this section only applies to the data path between GB to GB end points. The flow control between the MPI and GB device is maintained in the MPI-GB ADI implementation. For example, the MPI application can send back-to-back packets with MPI_Isend until the shared buffers are used up to the maximum threshold value defined by ADI. The MPI application can be unblocked when the GB or device driver (with DMA) consumes these buffers.

3.3.8 Multicasting

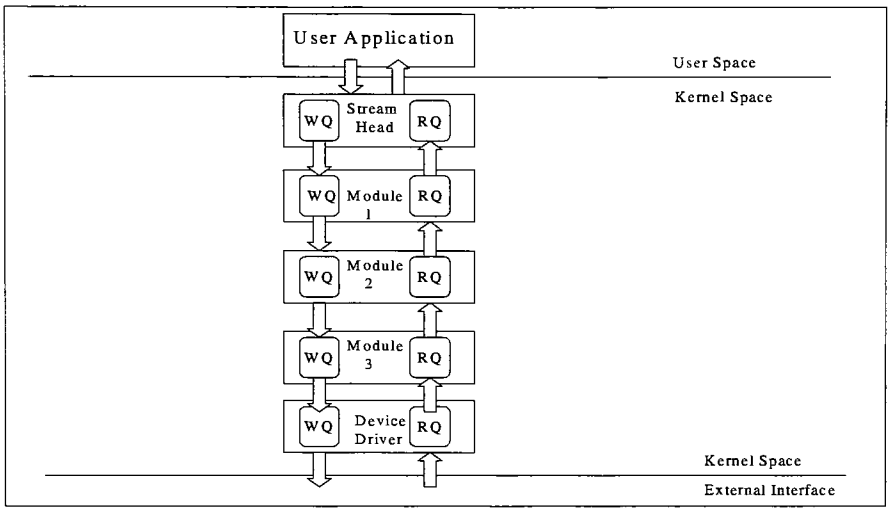
In MPICH, collective operations like ***MPI_Bcast*** are implemented in terms of point-to-point primitives. This means that MPI broadcasts require a separate ***MPI_send*** operation to all receivers. However this operation can easily be hastened by relying on the multicasting capability of most network devices.

In our implementation we use a FDDI card that has 32 entry Content Addressable Memory (CAM) Registers for multicast addresses. In the initialization time for each open connection, a pre-selected, single multicast address is programmed into every node in MPI communication world. In the MPI API, **MPI_Bcast** is updated to call **GB_Bcast** directly. **GB_Bcast** generates the appropriate FDDI header by inserting the multicast address into the destination field before handing the packet to the device driver.

3.3.9 Mmap vs Streams

STREAMS provides a framework for writing device drivers and network protocols. It enables a high degree of configurability and modularity. It allows writing independent modules, each of which acts as a filter and performs some specific processing on the data stream. It then allows users to combine these modules in different ways to form a stream. The stream acts like a bi-directional pipe, moving data between the application, the device and the network interface, with appropriate processing in between. This modular design allows network protocols to be implemented in a layered manner, each layer contained in a separate module.

Figure 27: *UNIX System V Streams Interface*



Most commercial network device drivers are implemented utilizing Streams on most of the UNIX variants. This gives NIC vendors a generic operating system device driver programming interface for device driver development. Even for non-streams based Operating System vendors (Windows NT, Linux, QNX , VxWorks) or third party service providers (Gcom) supply an artificial streams library is for streams driver/module development. More information for Streams can be found in [37].

The raw IP socket and DLPI models of MPIGB are developed utilizing the streams model. During the implementation and analysis of MPI-GB, we have faced the following difficulties:

- Solaris Streams scheduler:

  One of the main goals of this thesis is to show that it is possible to bring the valuable features of a network interface like deterministic worst-case latency to a distributed user application. Our timing analysis breakdown shows that the variance of worst case latency is mostly due to the high dependency of Solaris "streams" scheduler's scheduling time to load on the system. Here is how the Solaris scheduler works:

  When the streams queue ***put*** procedure defers the processing of data, it calls ***putq( )*** to place the data into the queue and then calls ***qenable( )*** to schedule the queue for servicing. ***qenable( )*** system routine sets the ***QENAB*** flag for the queue and inserts this queue to the tail of the list of queues waiting to be scheduled. Streams scheduler is implemented by a routine called ***runqueues( )***. The kernel calls ***runqueues( )*** whenever a process tries to perform an operation on a stream. This routine checks if any streams need to be scheduled. If so, it calls ***queuerun( )*** , which scans the scheduler list, and calls the service procedure of each queue on it. The service procedure must try to process all the messages on the queue. The kernel guarantees that all scheduled service procedures will run before returning into the user mode. In a high load, Solaris streams mechanism does not provide any prioritization of service queue scheduling.

  Even in Solaris manuals [22] it is indicated that the quanta vary from 20 ms to 200 ms; this of course can increase latencies in MPI-GB protocol and nullify all the good work that has been done. It is well known that Solaris is not a real time

operating system and STREAMS framework did not provide us a framework that can be customized for this specific use.

- User <-> Kernel Memory Copy

  A process writes data to a stream using ***write*** or ***putmsg*** system calls. The stream head copies the message from user space to streams buffers. On the other end, upon the ***getmsg*** or ***read*** call from the process, if the data is already available on the stream head, the kernel extracts it from the message, copies it into user space. (Figure 29) In both cases, the data is copied between user space and kernel space. In this thesis we intended to minimize the overhead including any data copies. The third version of MPI-GB utilizes the user to kernel memory map model to achieve this goal. (Figure 30)

3.3.10 MPI-GB ADI Primitives

All MPI and GB primitives and descriptions implemented or inherited from MPI-CH implementation is shown below in table 9. The ADI primitives regarding ready and send mode are not supported in MPI-GB implementation in this version.

Table 9: MPI-GB API

MPI-GB Primitives **Core Message Passing ADI Routines**	**EXPLANATION**
Send Routines	
MPID_GB_Post_send	Starts a send operation
MPID_GB_Post_send_Complete	Checks for a pending oper.
MPID_GB_Post_send_ready	Not Implemented
MPID_GB_Post_send_sync	Not Implemented
MPID_GB_Test_send	Tests for the completion of send request
Receive Routines	
MPID_GB_Post_recv	Posts a receive request
MPID_GB_Complete_recv	Completes a receive request
MPID_GB_Test_recv	Test for a completion of receive
Other Core Message Passing Routines	
MPID_GB_INIT	Initializes the ADI
MPID_GB_Iprobe	Checks if a specific msg have arrived
MPID_GB_Myrank	Rank of calling process
MPID_GB_Mysize	Number of processes
MPID_GB_Cancel	Cancels a pending operation Not Implemented
MPID_GB_End	Terminates the GB
MPID_GB_Clr_Completed	Not Implemented
MPID_GB_Set_Completed	Not Implemented
_MPID_GB_Check_device	Checks for a pending device operation and handles if found
MPID_GB_Commit	Addition to ADI implementation for Guaranteed delivery. Explained in separate section
Point2Point Extension Routines	
MPID_GB_Blocking_recv	Performs a blocking receive, Calls a post_receive, then complete_send.
MPID_GB_Blocking_send	Performs a blocking send, calls post_send and complete_send.
MPID_GB_Blocking_send_ready	Not Implemented
MPID_GB_Probe	Returns the status for a matching message Waits if a message not available.
Collective Extension Routines	
MPID_GB_Barrier	Performs Barrier Synchronization. Calls GB_sync
MPID_GB_Comm_free	Frees the ADI's use of communicator.
MPID_GB_Comm_init	Initializes the ADI's use of communicator.

MPID_GB_Reduce_sum_double	Not Implemented
MPID_GB_Reduce_sum_init	Not Implemented
MPID_GB_Bcast	Broadcast (to All) Calls GB_bcast
Environment Routines	
MPID_GB_Node_name	Provides the name of the processor
MPID_GB_Version_name	Provides version number
MPID_GB_Wtime	Returns the time, relative to some arbitrary point
MPID_GB_Wtick	Returns the resolution of the timer

Table 10: *GB API*

GB API	EXPLANATION
GB_Init	Initializes the GB
GB_Commit	Commits buffers, payload for guaranteed delivery
GB_Terminate	Closes all the connections. All context records are marked as NULL state.
GB_Send	Performs the send operation, Formats the data and control packets. It prepares the GB header for each packet and passes the MPI, GB, FDDI headers and data to network protocol interface.
GB_Poll	Polls for a receive packet
GB_Map	Maps the user buffers to kernel memory used by GB_Commit.
GB_Unmap	Unmaps the buffers mapped bt GB_Map
GB_SetHandler	Add new entry to handler table for receive packets.
GB_Receive	Receiver acts on packets received from network. It identifies the message and calls the appropriate handler based on message index value. This routine is not called directly from MPI
GB_Bcast	Broadcast (to All) Cal
GB_Version	Returns the version string of MPI

3.3.11 Three MPI-GB Solutions

Keeping the issues such as simplicity and portability in mind, the first model, a raw socket streams implementation of MPI-GB, is designed to interface with IP via raw sockets. This allowed us to remove the overhead of the p4 and TCP protocols. The second version took this approach one step further. In this version, the streams DLPI implementation of MPI-GB is designed to interface with the FDDI device driver via DLPI and streams calls. Although issues such as implementing efficient flow control, buffering, startup, discussed in chapter 1 are addressed in both of these first two versions, other issues such as avoiding context switching copying data from/to user space are not considered in these versions. Only a few modifications to the FDDI streams device driver were necessary in these first two implementations.

The latest version of MPI-GB implemented to date was designed by modifying the first two versions. The GB and device driver interface DLPI streams framework is replaced with flat MMAP user interface. This job included adding an MMAP kernel interface FDDI to the device driver to avoid redundant data movement between kernel and user space. We integrated our own scheduler workaround to avoid the non-deterministic Solaris streams scheduler. We also added support for payload request primitives to MPI.

This incremental design and development approach not only made the design and development easier, but also helped us to measure the impact of each version.

3.3.11.1 Streams Raw Fast IP Socket Interface Model

Incremental steps were taken to design and develop the device interface of the MPI-GB protocol. For the first version full attention was given to develop the GB protocol and

MPI-GB ADI interface; any device driver interface modifications were left to the later versions.

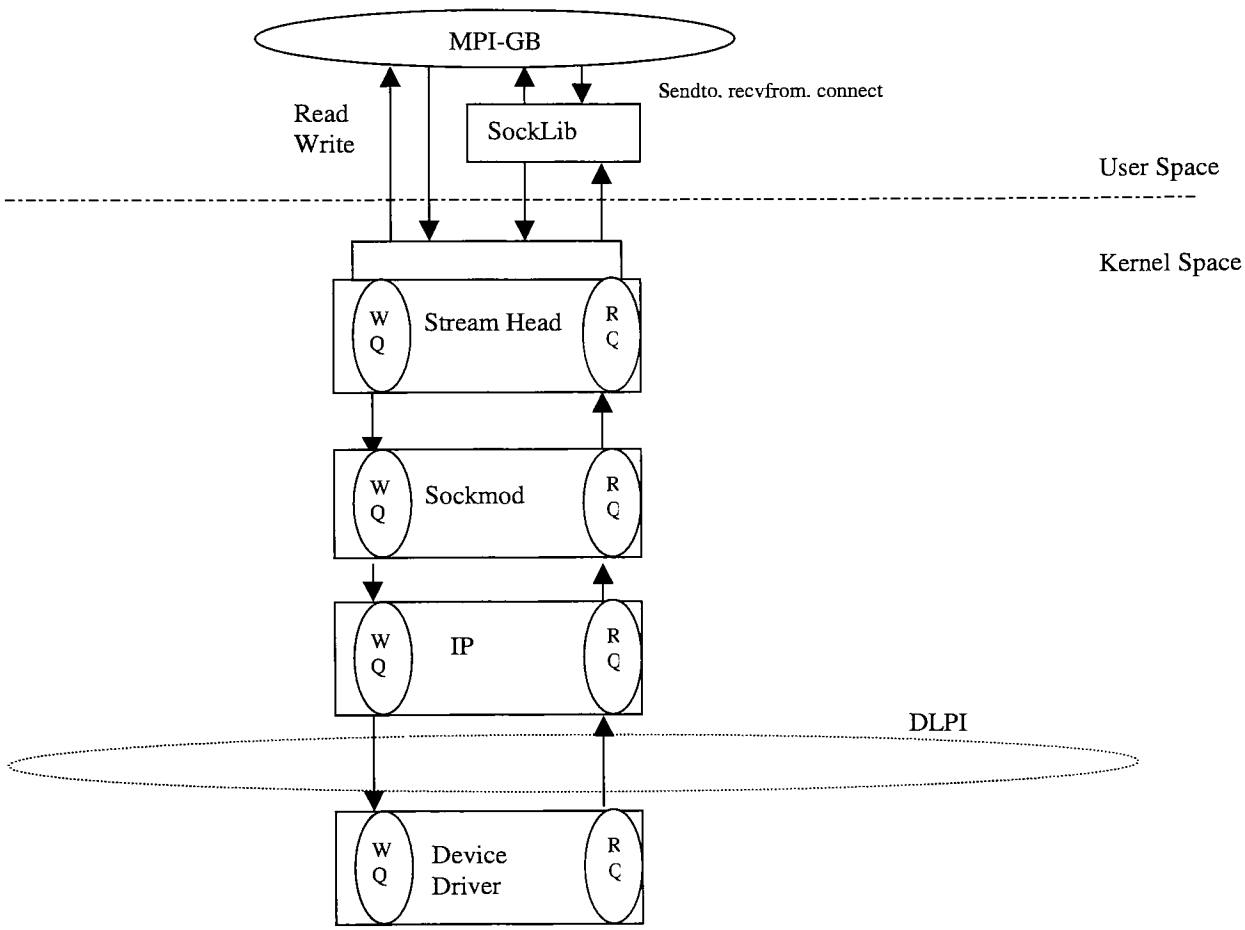
For this version, we use the existing framework provided by UNIX systems. One option was to develop GB on top of UDP utilizing UDP sockets for the interface among them. GB itself is a transport protocol; in order to avoid having two-transport layer protocols, GB was designed to interface with IP via raw sockets. (Figure 28)

Raw Sockets provide powerful features such as allowing the user to build his/her own IP header. (With ***IP_HDRINCL*** option set in ***setsockopt*** call)
The GB send and receive handler routines can read and write IP datagrams using the Ipv4 field that is not processed by the kernel when utilizing raw sockets. This field is set to 99 for the GB protocol. This value is provided as a third argument to *socket* call.

```
int  sockfd;
int protocol=99;
sockfd = socket(AF_INET, SOCK_RAW, protocol);
```

By using raw sockets, solving the fragmentation and re-assembly problem is deferred to later revisions, since IP (Module in Solaris kernel) fragments the raw packets that exceed the outgoing interface MTU (4352 for FDDI).

Figure 28: *Raw IP Socket implementation of MPI-GB*



Utilizing raw sockets and IP simplified the design process. It allowed us to test and use the GB protocol in existing public systems with no extra requirements because it utilizes publicly available raw sockets and IP.

The major performance bottlenecks such as context switching, data copying from/to user and kernel space are not addressed in this version. Actually, this version performed better than the MPI-CHP4 implementation (Figure 31). Along with general bottlenecks

mentioned above, this version separated the transport protocol (GB runs in user space) from the network IP protocol (runs in kernel space) with sockets API. Therefore this approach introduced an extra overhead due to the additional communication between user and kernel processes. However it provided a good stable framework for testing the GB protocol and MPI-GB without introducing more problems.

3.3.11.2 Streams DLPI Raw Interface Model

SVR4 provides datalink access through DLPI. Data Link Provider Interface, DLPI is a protocol independent interface designed by AT&T that interfaces to the service provided by the datalink layer. Access to DLPI is by sending and receiving streams messages. (***getmsg, putmsg, read, write***)

To tap into the datalink layer, GB opens the device and attaches and binds to it using DLPI requests. All requests are actually streams ioctl requests with DLPI commands. GB configures the DLPI stream as "RAW" so that it can send and receive the FDDI header to/from device driver.

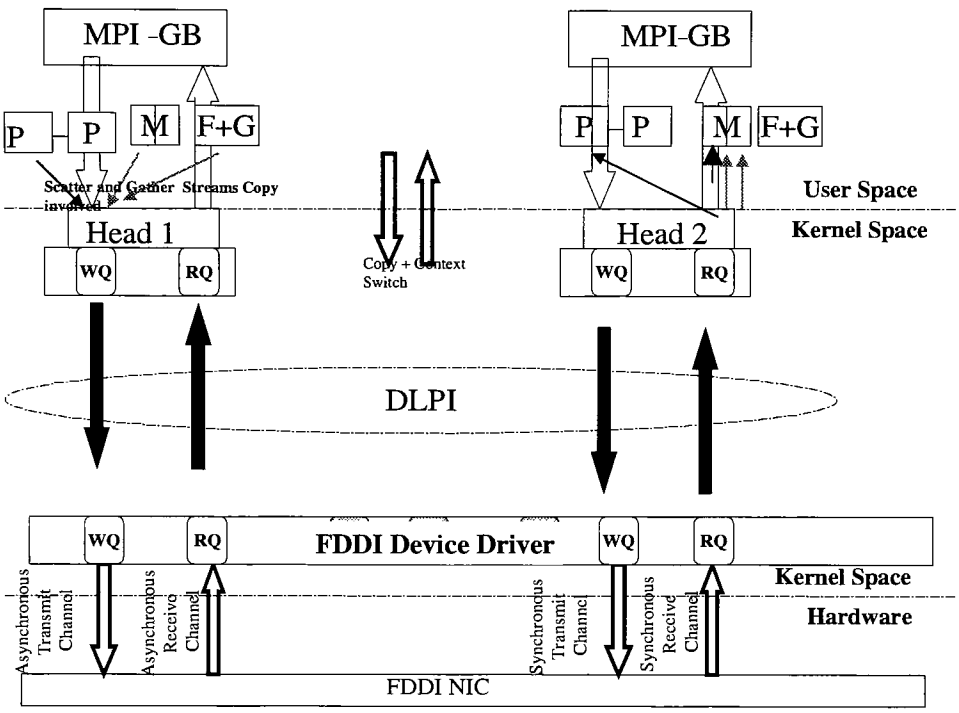
Direct interfacing with devices utilizing DLPI commands helped in eliminating the overhead caused by the socket library and IP. But it also introduced the segmentation and re-assembly problem that needs to be solved for packets larger than MTU size. In this version, GB acts a transport and network layer protocol. When the ***GB_send*** routine is invoked with MPI header and pointer to the data and length, it splits the data buffer into several fragments. It constructs FDDI and GB headers for each fragment. It marks the ***more fragment flag*** in GB headers except for the last one. Splitting and header insertion operations are performed during enqueing the message into the write queue. The splitting

operation does not require temporary copy buffers becuase the pointers to the data buffers are increased by (MTU-60) every time before inserting the packets into the queue.

The driver extracts the packets from the write queue, prepares the transmit descriptors and starts the DMA operation. The receive operation is a mirror operation to transmit. As the packets are received in GB, data fragments are assembled in the receive buffer. Once the packet is received with GB, it only extracts the header. The GB header carries an index into handler tables. The corresponding handler function is invoked. This function looks into posted queue to find the pointer to the receive buffer. It extracts the remaining data part into the receive buffer, then marks the *receive* state as complete if the more fragment flag is not set. If it is set, it waits until all fragments are received before marking the receive state.

The roundtrip latency performance of the DLPI model presented a significant improvement over the Raw IP socket implementation. (Figure 31). For all packet sizes, it provides much better roundtrip latency than other MPI-CH implementations. Although many copy overheads are eliminated in the implementation, there is still room for improvement in the overhead caused by user/to kernel context switching, streams overhead for short messages, and data movement for long messages.

Figure 29: Streams DLPI interface model implementation



3.3.11.3 MMAP user interface Model

During the analysis of the first two implementations, it has been observed that for small messages, system call overhead and streams interface overhead dominate the total time required to transmit messages. While for large messages, data movement is the main factors which determines the communication performance.

In the MMAP implementation, one goal was to eliminate the remaining copy between user and kernel processes (for larger messages). Another objective was to avoid the streams overhead and context switching in order to improve the communication performance. (for small messages)

It would be best if we have unlimited memory on the board and to have this memory be accessible and pre-mapped to user space and kernel space. In that case data never needs to travel on the I/O-to-memory bus until accessed by the application. Unfortunately this kind of hardware support is not provided by the FDDI hardware used in this thesis.

Instead, the MMAP module employed in this model lets the kernel and user to manage the interface memory that resides in the device driver. It defines a new framework with shared memory between the user and kernel spaces, and uses a Scatter Gather DMA to move the data between the shared memory and the network interface.

Unfortunately the descriptor management logic and DMA engine in FDDI ASIC requires 4K page aligned buffers to be given by the host before the packet is received. This requirement makes it impossible to implement the true zero copy model. In our model, to overcome to this limitation, the simulation is performed so that all the receive buffers are quad word (fastcopy) copied from user-kernel shared memory, therefore meeting the DMA engine's requirements. So in reality only one copy is performed in the user space to overcome this hardware limitation.

This model uses transmit buffer and the image of receive buffer pool that is pre-mapped in both kernel and user space at initialization time. This is done with ***GB_Commit***, ***GB_map*** commands. Solaris ***mmap*** command is responsible for registering and mapping the kernel buffers.

This copy avoidance technique requires a close cooperation between the application, networking software, and device driver, all allocating memory from the same pool. Implementing this model, required us to make major changes in the GB device interface and device driver. The device driver is modified to work, as a mmap device driver since streams framework is no longer employed in the data transfer.

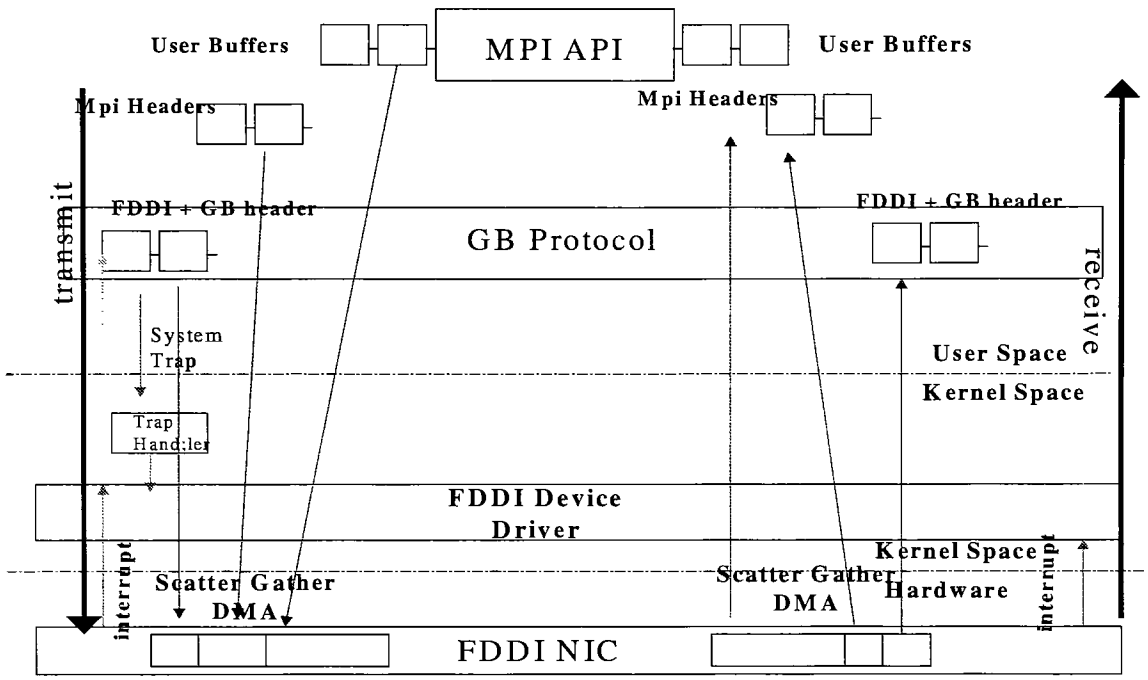
To minimize the scheduling and context switching time overhead caused by streams, trap and trap handler routines are implemented. GB processes are switched to run in real-time mode in order to give priority over system and time-shared processes.

Another improvement is made in process control by locking the critical areas of the GB process address range with ***memcntl*** system call. This prevents the GB process pages from being stolen by the system memory management code in case of high load. This is very important, since if the page must be brought from secondary storage, it will increase response time dramatically.

Roundtrip latency and bandwidth results are more or less the same as the results recorded with DLPI implementation during the idle asynchronous load. (Figure 31). When the asynchronous network and system load injected by ttcp, this MMAP and DLPI implementation outperformed other implementations. (Figure 35)

MMAP and DLPI implementations (Figure 34) helped in decreasing the number of the packets missing the deadlines with MMAP and DLPI implementations (Figure 34). This improvement is attributed to setting the GB process to a real-time priority and as well as memory management improvements.

Figure 30: *MMAP model of MPI-GB*



4. Performance Analysis of MPI-GB

This chapter provides measurements of the performance of MPI-GB. All benchmark results are recorded on a FDDI network cluster of SPARC and X86 workstations running Solaris OS. As these results are collected while the MPI-GB project is in progress, we were able to analyze the effect of each revision and major modifications. The performance of each version will be contrasted with the performance relative to the performance of other MPI implementations.

4.1 Test Conditions and Measurement model

All tests documented in this chapter and chapter 1 were executed using the workstations in table 12. ("Bench" Test Bed). The Benchmarks listed in Chapter 2 were run on a FDDI network of workstations listed in "PTI" test bed (Table 11)

Tables below lists some hardware characteristics of the machines we used in two different test beds.

***Table 11:** "PTI" Test Bed Workstations used in this thesis -*

CPU Type Speed	Cache Size	Memory	Page size	Bus type Speed	Solaris OS Version
Ultra-1 168 Mhz	256KB	128 Mbyte	8192	PCI 33Mhz	2.5.1
Ultra-10 300 Mhz	512KB	128 Mbyte	8192	PCI 33 Mhz	2.6
Ultra-1 168 Mhz	256KB	128 Mbyte	8192	SBUS 25 Mhz	2.5.1
Sparc Station V (sun4m) 80 Mhz	64KB	64 Mbyte	4096	SBUS 25 Mhz	2.5.1
Pentium 166 Mhz	256KB	32 MByte	4096	PCI 33 Mhz	2.6 X86

Table 12: *"Bench" Test Bed - Workstations used in this thesis -*

CPU Type Speed	Cache Size	Memory	Page size	Bus type Speed	Solaris OS Version
Ultra-1 168 Mhz	256KB	128 Mbyte	8192	PCI 33Mhz	2.6
Ultra-5 300 Mhz	256KB	128 Mbyte	8192	PCI 33Mhz	2.6
Pentium 233 Mhz	256KB	64 Mbyte	4096	PCI 33 Mhz	2.6 X86
Pentium 166 Mhz	256KB	64 MByte	4096	PCI 33 Mhz	2.6 X86

The purpose of this thesis is to show that it is possible to bring the features of a network interface with high performance, low average and worst-case latency and guaranteed bandwidth to a distributed user application. To serve this purpose, these benchmarks will evaluate the performance at the application level. At this level both the network and software overhead can be measured. Therefore, most benchmarks were executed at the MPI application level. Each benchmark consists of two or more processes. As the current versions of GB do not allow more than one process multiplexing on a receive/transmit site, only one process can be executed on each workstation. MPI application test codes used to measure the performance in this thesis are included in the Appendix.

For most of the benchmarks (Except the Worst case delay benchmark which is for determining the quality of guaranteed bandwidth delivery to the application), we made sure that no other network application or daemon is using the network at this instant. These experiments are performed on the Solaris OS platform, which is a time-shared multitasking system. Each measurement is repeated a number of times, and a median

time is taken, to filter out spurious results due to the CPU load surges, scheduling problems which may be arising due to the other processes time sharing the CPU.

Most of the benchmarks were executed on workstation pairs using the "point-to-point" send/receive operations. In addition to these benchmarks, a separate benchmark is developed to evaluate the performance of the MPI-GB collective communication primitive, MPI Broadcast. As explained in the previous chapter, MPI-GB implementation utilizes FDDI device specific operations to support collective communications. The MPI-GB implementation is obviously superior to MPICH-P4 implementations, as the MPICH implementation utilizes a collection of point-to-point operations to support collective communications.

We made sure that the "receive" requests are always posted before the message arrives. (There is no need for an "unexpected queue."). First trials are always ignored to avoid any communication setup time.

4.2 Performance Metrics

The message transmission time to send a packet from the receiver to the transmitter can be decomposed coarsely into three steps.

1. **Send Time:** The sender has to spend time to do some packet processing, such as message copying, and packetizing. This latency may be called as "sending latency" :$\mathbf{T_{send}}$. For MPI-GB, this time consists of the combination of time spent in moving data from MPI application to GB, from GB to device driver, from device driver to NIC. This

processing time in each layer will be included in the moving time.

$$T_{Send} = T_{MPI} + T_{GB} + T_{D.Driver}$$

2. Network Latency: After being put onto the network, the message has to spend some propagation delay in the network before it reaches to the destination. This latency may be called as "Network Latency": $\mathbf{T_{Net}}$. In case of synchronous traffic, this latency is constant.

$$T_{Net} = C_{FDDI}$$

3. Receive Time: Once the message arrives, the receiver host picks up the message from the network device and perform some packet processing, such as message copying, reassembling, and header extracting. This latency is called "Receiving Latency": $\mathbf{T_{Rec}}$

$$T_{rev} = T_{MPI} + T_{GB} + T_{D.Driver}$$

Comparing the different versions of MPI-GB with other MPI implementations requires measuring several metrics. The following three metrics were chosen to evaluate the performance of MPI-GB for its point-to-point communication primitives:

1- Communication Latency (T_c)

The communication latency is defined as the time that a process has to spend when it sends and/or receives a message. This time includes constant start-up latency, $\mathbf{T_s}$, which includes the fixed cost of system call and the initialization overhead. The communication latency is also proportional to the message size (n).

$T_c = T_s + n* T_{Net} + [H+n/MTU]*T_p$

($T_{Net} << Ts$)

T_{Net}: Fixed transmission network latency for one byte.

T_p: Packetizing delay proportional to the message size. (Data copying, assembling/disassembling)

H: Number of headers.

MTU: Maximum Transmit Unit

The Effect of startup latency to communication latency is reduced when n increased as ($n* T_t$) increases.

2- Roundtrip Delay (T_{end})

The Roundtrip (Ping-pong) delay is the communication latency, which equals a complete roundtrip message iteration. It equals to the sum of the time taken by the process to send a message to the other process and the time it has to wait for a reply from that process. This latency includes both the network and software overhead. A Ping-Pong delay test is provided to evaluate this metric for MPI-GB.

$$T_{end} = T_{send} + T_{rev} + 2*T_{net}$$

3- Bandwidth

Bandwidth is the rate at which the network can deliver data. The bandwidth can be directly computed from the communication latency by

$$B = n/(T_c \times 10^6) \text{ or } B= 10^{-6}/(T_s/n + T_{Net} + T_p/MTU),$$

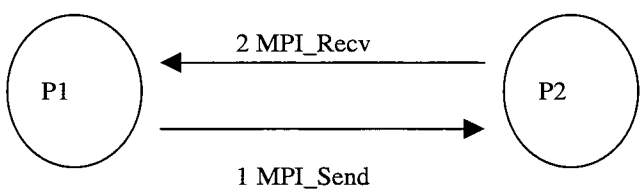
As seen from the formula, when the message size increases, the effect of start-up latency to bandwidth decreases.

4.3 Point to Point Benchmark programs

Points to point to benchmark tests are provided to measure and compare the effective bandwidth and latency of MPI-GB versions relative to the performance of other MPI implementations.

4.3.1 Roundtrip (Ping-Pong) Delay measurements

The Roundtrip Delay measurement test is an MPI program, which measures the total time to send a message and time taken for the message to return. The communication primitives used are MPI_Send, MPI_Recv. As we are interested in the roundtrip delay, blocking send and receive is used.

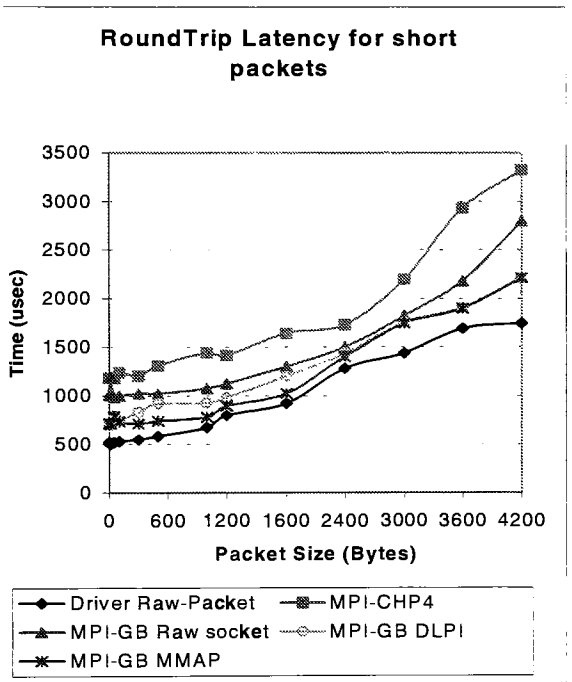


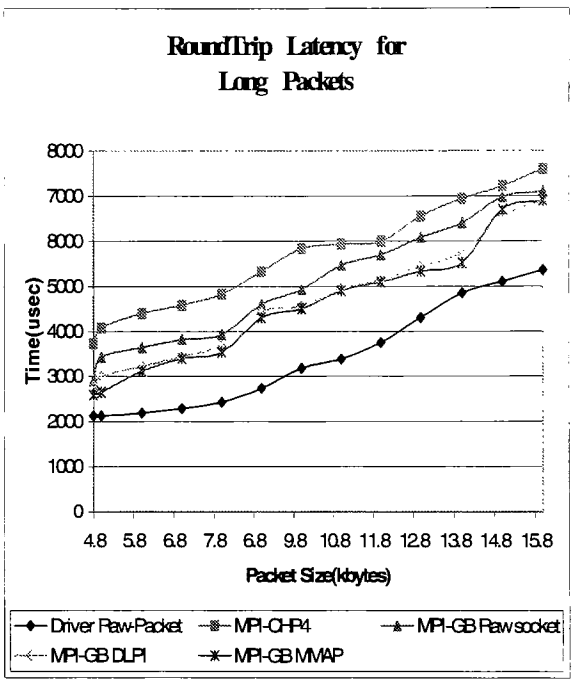
The roundtrip latency performance results of three MPI-GB implementations are compared to the MPICH-P4 base implementation and raw data transfer from/to FDDI device driver and are given in Figure 31. The roundtrip latency results of MPI-GB versions present an incremental improvement over the previous versions. As shown in the figure, all three implementations outperform the MPI-CHP4 implementation for all message sizes.

For short messages, the MPI-DLPI and MMAP roundtrip latency results are very close to the raw data transfer results obtained from device driver.

The difference between the raw data transfer and MPI implementations for long packets can be attributed to the extra copy done to satisfy hardware's DMA engine requirements.

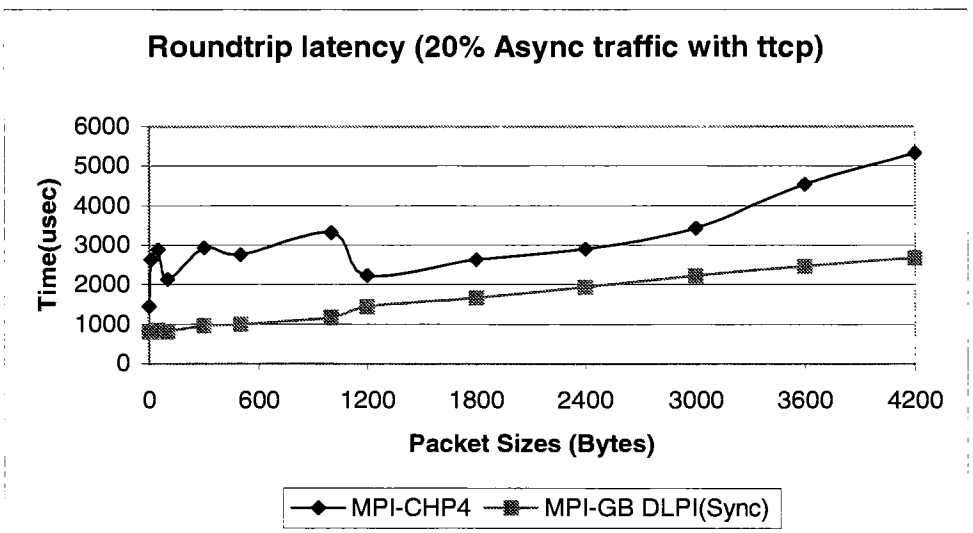
Figure 31: *Round Trip latency of MPI-GB for Short and Long packets*





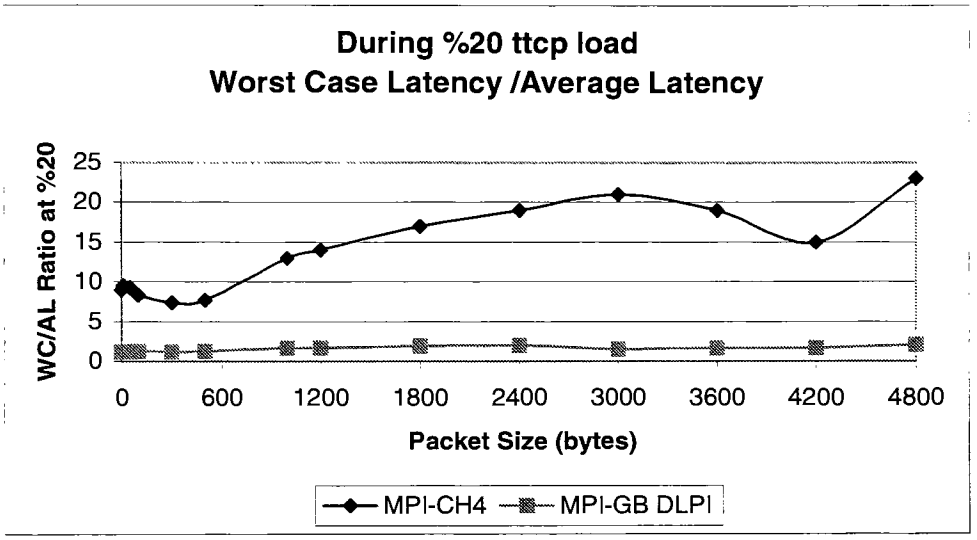
Roundtrip latency measurement for our MPI-GB DLPI version and MPI-CHP4 are repeated when an asynchronous (20Mbit/sec) TCP/IP load is injected into the FDDI ring with **ttcp**[35] benchmarking program. The results are given in Figure 32. Given the packet prioritization, scheduling and redundant copy improvements, MPI-GB performs deterministically during the high asynchronous load.

Figure 32: *Roundtrip latency for MPICH-P4 and MPI-GB during 20Mbit/sec ttcp load generated at the workstations.*



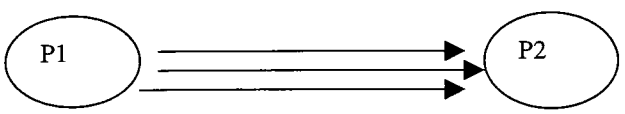
We also measured the worst roundtrip time for each message length. The ratio of worst message delivery time to average message delivery time presents a good performance metric for real-time traffic. As seen from the Figure 33, this ratio ranges from (1-2) for MPI-GB whereas the ratio is anywhere between 7 to 22 for MPI-CHP4.

Figure 33: Worst case Latency/Average Latency ratio during 20 Mbit/sec traffic



4.3.2 Bandwidth Measurements

The bandwidth test is a MPI program, which measures the time to send a sequence of messages back-to-back from one node to another. The sender keeps sending data unless it is blocked (no more free buffers between the GB device and MPI), and the receiver keeps consuming the data.



The communication primitives used are MPI_Send or MPI_ISend, MPI_Recv, MPI_Irecv, and MPI_Waitall. Nonblocking primitives are used since pipelining the

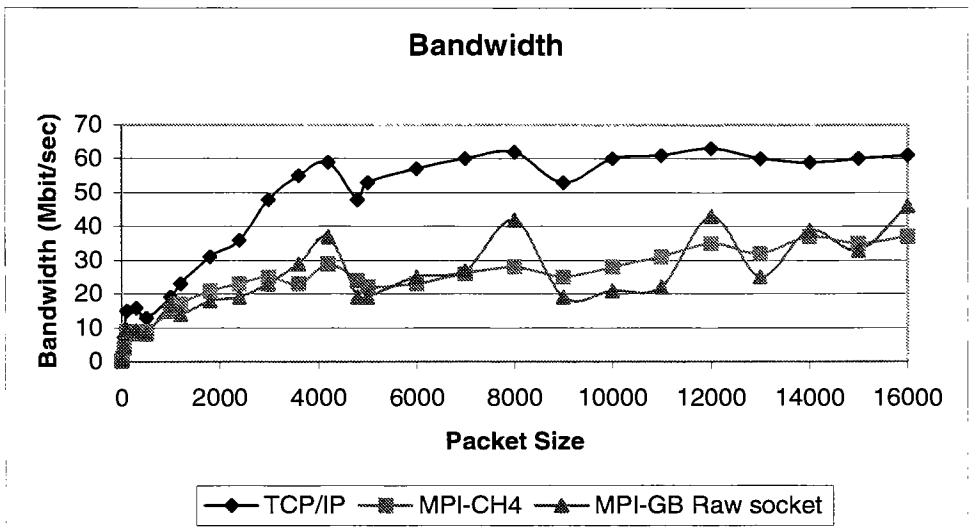
messages for send and receive is important in the bandwidth test. (Try to send and receive as many as possible).

This test results can be limited by a bottleneck on either the receive or send sites. It has been observed that the receiver needs to send a negative acknowledge to slow down the sender. This is the case when the receiver is slower than the sender. MPI-GB interface buffers are filled up before the MPI application can consume them. Until the negative acknowledgment packet reaches to the sender, the sender continues sending packets; this may cause packet loss at the receiver site. If the receiver detects an out-of-order sequence number, it asks for a re-transmission. Occasionally, this buffer is found overwritten on the sender site, and the sender cannot fulfill this request and the application fails. Due to time limitations, we are not going to solve this problem by providing better rate and flow control algorithms. One possibility is to allow the network interface to reject the packet and resend it. This and other limitations of MPI-GB protocol, and how they can be addressed in future versions will be presented in Chapter 5. Another interesting thing is that this failure is mostly observed in the second and third versions of MPI-GB. Since packet copying and context switching are eliminated, the send routine of this version is faster than the send routine of the Raw IP socket version of MPI-GB.

Figure 34 presents the MPI-GB raw sockets and MPICH-P4 bandwidth measurements for different message sizes. MPI-GB bandwidth measurements were inconsistent due to the reason explained above. We used two separate thresholds; One threshold is used to

decide when to use negative acknowledgment and the other one is used to determine how much the sender needs to be slowed down affects the performance.

Figure 34: *MPI-GB, TCP/IP, MPI-CHP4 bandwidth for different packet sizes.*



4.3.3 Percentage of missing deadlines packets.

In this section, the main feature of GB protocol, the guaranteed bandwidth delivery to the application, is evaluated. The metric used for this evaluation is the percent of frames that missed their deadlines.

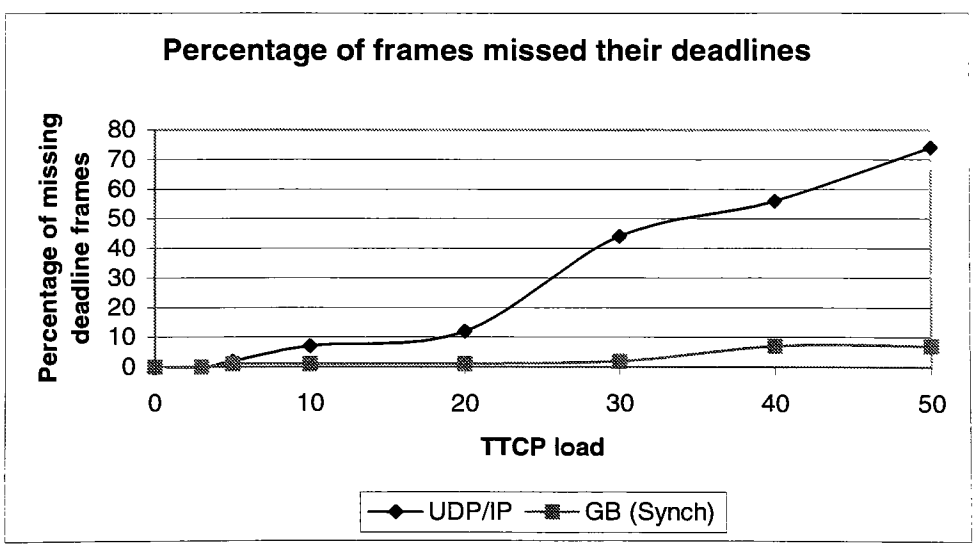
Most real-time applications currently use UDP/IP along with RTP to deliver real-time traffic. In this section, GB itself is evaluated against UDP with respect to the percentage of frames that missed their deadlines.

Using the test network described in Table 12, The test program in the appendix is executed to exercise the GB directly. The 4000-byte synchronous frames were sent every 100 msec. The packet is considered to be overdue when it cannot be received in 105msec time frame. (100msec+5msec (Grace period))

The percentage of packets missing the deadlines are recorded on the receive site as the asynchronous network load (extra UDP/IP load created with TTCP[35]) increased in each step.

The actual payload needed is 0.32 Mbit/sec for the network. The payload is programmed with MPI_GB_Commit for 10 Mbit/sec to compensate for any host software delays. This is equivalent to 157 synchronous units.(Table 9). The SBA state machine is programmed with this 157 synchronous units parameter.

Figure 35: *Percentage of frames, which missed their deadlines*



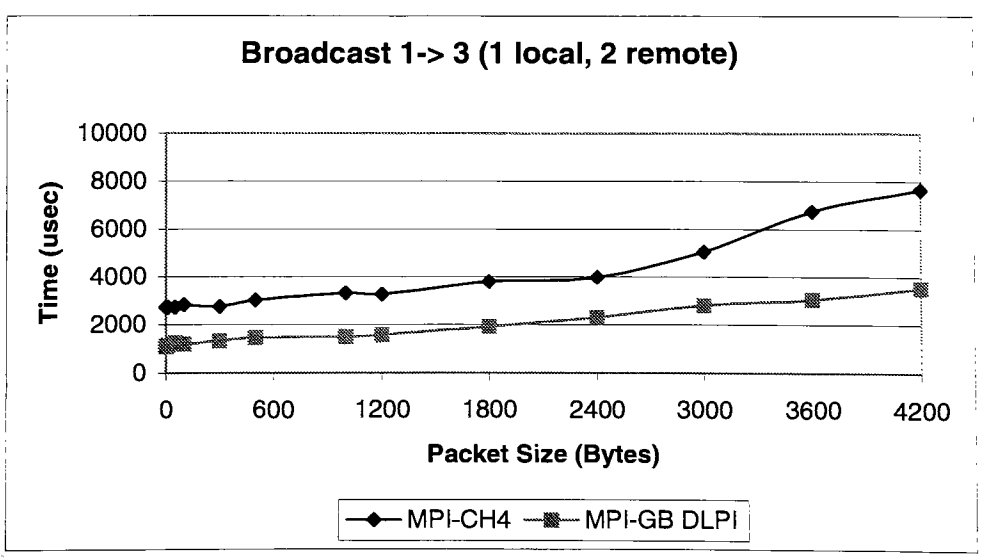
The results are recorded with GB and UCP/IP [Figure 35]. Under a 50Mbit/sec asynchronous load, only 7% percent of the message missed their 5msec deadlines using GB. On the other end 76% of the UCP/IP packets missed their deadlines or were not received at all.

4.4 Collective Communication (Broadcast) Measurements

The primitive chosen for collective testing is the "broadcast". This is the most commonly used primitive in parallel programming. This primitive simultaneously sends the message to all members of the group. The MPI-GB provides a device specific multicasting mechanism (multicast addressing) to support this call.

The performance of MPI DLPI version broadcast outperforms the performance of MPI-CH implementation. This is primarily due to the fact that MPI-CH uses point-to-point operations (n sends for n stations) as MPI-GB uses one send to broadcast to n stations. (See Figure 36)

Figure 36: *The time of MPI_Bcast as function of message sizes.*



4.5 Real World Example: Distributed Ray Tracing

Distributed Ray Tracing, utilizing PVM, was implemented as a class project prior to this thesis. In this project, the processor farm method was utilized. The jobs were distributed to the available slaves on a line granularity basis. During the analysis of this project, we observed that the time taken for each line (job) completion was directly proportional to the load of network and CPUs. Although the computation time is nearly equal for each line, the network packet delivery time (total communication latency) is largely dependent on the network (collision in Ethernet) and packetizing load (PVM, TCP/IP and Ethernet device drivers)

As a part of this thesis, the Distributed Ray Tracing application is ported to work with MPI implementations. It has been tested with our DLPI MPI-GB and MPI-CHP4 MPI implementations. The packets are marked as synchronous by the MPI-GB implementation for prioritization and guaranteed delivery. While tests are running, 20 Mbit/sec tcp/ip load is injected to FDDI network from the workstations where Ray Tracer slaves are running. Distributed Ray Tracing code is tested with PVM, MPI-CHP4 and MPI-GB with the same initial dataset and three slaves.

The PVM version performed slightly better than MPI-CHP4 version (completion time of 298 sec vs 318 sec). On the other hand, MPI-GB outperformed the others; and all the jobs are completed in 219 sec.

5. Conclusions

During the last decade, many new communication technologies (ATM [19], FDDI [6], Fiber channel [28]) have been developed and employed to bring high bandwidth, low latency and guaranteed bandwidth to the network interface. User applications can benefit from these technologies only with carefully designed protocols.

By designing and implementing the MPI-GB protocol, we showed that it is possible to get the user application benefit from the features that network interface provides, without losing much in reliability.

5.1 Accomplishments and Summary of Study

In the analysis phase, existing programming environments, standards, protocols and recent research have been investigated thoroughly. During this investigation, valuable techniques for solving the major protocol problems such as connection, flow, error, rate control, buffer management etc. were learned from the literature: Fast Sockets, FM [8], AM [27] and fbufs [31] and many others. As we tackled these problems in our design, the techniques described in these references were evaluated and adapted to suit our needs.

For this investigation, the MPI-GB protocol was designed and implemented with incremental steps to meet the demanding needs of mission critical distributed applications. In the design, two main concepts are articulated: 1) Being given a very reliable underlying network, extra error checking should be removed from the protocol to achieve a minimal protocol overhead. 2) Using real time schedulers, priority queuing, and removing extra copying, the gap between user applications and the network hardware

should be narrowed. GB protocol is designed to provide a transparent transport and network protocol with the high performance, low latency and guaranteed delivery of FDDI to the user application.

The extra protocol overhead, buffering and inefficiencies arising at the interface between these layers at the receive and transmit sites are removed. Special care is given for scheduling between layers and passing data between layers. Finally static buffering, flow control, efficient fragmentation and re-assembly features are carefully integrated into MPI-GB for a reliable and efficient data transfer. For example if the maximum threshold value is reached on the receive site, receive site sends a control packet to sender for it to slow down. Using this negative acknowledgement approach helps in eliminating the overhead caused by acknowledgment packets. An early threshold warning approach gives a sender an option to take precautions such as slowing down the transmission; this allows the receiver a chance to process and free more buffers before all receive buffers are depleted. For packets larger than the MTU size, fragmentation is performed with the help of "more fragments bit" on the GB header. On the receive end, packets are re-assembled in the posted receive buffers.

We chose the MPI-CH for a message-passing interface API for our protocol and developed an ADI utilizing the GB protocol. MPI-GB is developed by one-to-one mapping ADI functions with GB API. This allowed us to compare the performance of MPI-GB with other freely available implementations MPI.

Our MPI-GB implementation collapses the API, and the GB and device interfaces together. For example, the receive handler first extracts the MPI header from the streams/device memory, finds the handler function with the index from the handler table, searches the posted receive queue and assembles the packet directly into the posted receive buffer.

An efficient header handling mechanism is built into MPI-GB utilizing the FDDI hardware's scatter-gather DMA capability. Send packets are constructed at the network interface by gathering from three separate buffer locations.

The MPI-GB device driver interface is implemented in an incremental process of tackling the problems identified in the analysis phase. The inefficiencies arising from passing the data between the GB and the network interface was minimized step-by-step with each new version.

The first step was to design and develop MPIGB ADI and GB transport protocol on top of an already functional device driver and its API. Raw level Sockets and IP provided an excellent framework for this purpose. Although this version is not optimized, the performance was found to be slightly better than the MPI-CHP4 implementation. This version suffered from distinct layering costs of GB; sockets interface and IP increased user-kernel transitions on communication events. Nevertheless the roundtrip latency performance was slightly better than MPI-CH4.

To bring the MPI-API closer to FDDI hardware, DLPI version of MPI-GB is implemented. The GB device is redesigned to interface with a device driver with DLPI

streams calls. GB acts like a transport and network layer in this version. The Network protocol implementation issues such as segmentation and re-assembly, MAC vs. IP address are resolved with this version.

In an effort to collapse more communication layers and to avoid user-to-kernel data copying and context switching, the MMAP model is implemented. This implementation model defines a new framework with shared memory between the user and kernel spaces. It uses a Scatter Gather DMA to move the data between the shared memory and the network interface.

The other optimization attempted in this model is to minimize the scheduling time and to avoid the user context switching between user and kernel processes. Trap and trap handler routines are implemented for the send process to switch to kernel mode immediately. Another attempt was to switch the GB protocol process to a real-time process. A real-time process is given a higher precedence. It is guaranteed to be selected to run before any time-shared process.

This model required major changes in the device driver and GB interface. The shared memory managed model brought extreme complexity to the device driver. Also due to the receive FDDI-ASIC-DMA engine buffer requirements, the implementation is not a true zero copy model. However performing this copy in the user process with quad words, adding trap mechanism are the main optimizations to this model. This model can easily be modified to work as a true zero copy software if used with hardware with an enhanced DMA engine.

During high and idle asynchronous loads, MPI-GB and other MPI implementations are evaluated based on generic network evaluation metrics such as bandwidth, Ping-Pong (Roundtrip) latency and guaranteed bandwidth metrics such as worst case roundtrip latency and percentage of packets missing their deadline.

Our results indicate that MPI-GB provides much better roundtrip latency than MPI-CHP4 during idle load. The roundtrip latency results of MPI-GB versions present an incremental improvement over the previous versions. All three implementations outperform the MPI-CHP4 implementation for all message sizes.

For short messages, MPI-DLPI and MMAP roundtrip latency results are very close to the raw data transfer results obtained from the device driver. The difference between the raw data transfer and MPI implementations for long packets can be attributed to the extra copy done to satisfy hardware's DMA engine requirements.

MPI-GB primarily serves for guaranteed message delivery. Worst-case roundtrip message delivery latency is 1.5 times of average roundtrip message latency during the high load. This ratio was anywhere between 7 to 22 for MPI-CHP4.

GB itself is evaluated against UDP/IP during an external load injected using the TTCP program. Along with RTP, UDP can be considered as real time protocol. GB message miss ratio is only ~5% during the high asynchronous network load (50 Mbit/sec) whereas this value is close to 76% for UDP/IP packets during the same load.

This work shows the important issues involved in building the communication API for distributed applications. It proves that sharing a common information model between communication layers, and bringing communication layers closer minimizes software overheads by eliminating of redundant copies and context switches.

5.2 Future Work

MPI-GB implementation model provides an excellent framework for future study. A number of improvements can be made to GB protocol, network interface and MPI interface.

The GB protocol is designed and implemented for reliable networks such as FDDI. It relies on reliability and error control mechanisms of the underlying network. It does not provide a full error/flow control mechanism. Its flow/rate control mechanism is designed only to handle, and to recover from, buffer shortage problems on the both ends. (Receiver initiated slow down messages). Some cautions are taken such as retransmission of the packet with a receiver initiated negative acknowledgment in case a miss is detected in sequence numbers on the receive site. If the sender still has the buffer available, it restarts the transmission from that point. A few times during the testing bandwidth benchmarking with back-to-back packets, we found the transmission is failed because the buffer was overwritten by the sender. A future study is needed to implement a more reliable error/rate control mechanism that would not effect the performance negatively.

MPI-GB supports only the eager mode of MPI message passing mechanism. Eager mode means the data is delivered without waiting for the receiver to request it. We used MPI,

as an application programming interface to our GB protocol. The modes that do not require guaranteed delivery are not supported. But future implementations can modify MPI-GB to support the other modes: e.g. rendezvous mode [20]

Multiple processes are not supported by the GB protocol. Although a port number field is included in the GB header for possible multiple processes support; the current version of GB does not include multiplexing, de-multiplexing mechanism in the GB API.

The other limitations of this design involve the memory mapping mechanisms used in MPI-GB. Current "mmap" shared memory implementation assumes that data is accessed only in a single application domain. MPI-GB forces the application to register the send and receive buffers from the device driver (kernel) at initialization time. It restricts the application to predefine all receive and send buffers it will use in its lifetime. This restriction can be removed by using an efficient domain-independent buffering management mechanism like fbufs[31] . The fbufs mechanism combines two well-known techniques for transferring data across domains: shared memory, where page remapping dynamically changes the set of pages shared among a set of domains or using page remapping, where changes that have been mapped into a set of domains are cached for use by future transfers [31].

REFERENCES

[1] V.Sunderam, "PVM: A Framework for Parallel Distributed Computing." Concurrency, Practice and Experience, 2(4) pp. 315-340 (1990).

[2] Message Passing Interface Forum, "The MPI Message Passing Interface Standard Technical Report". University of Tennesse, Knoxville, (4/1994) Available http://www.mcs.anl.gov/mpi/mpi-report.ps.

[3] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface Forum", (7/1997).

[4] Message Passing Interface Forum,"Real-time message passing standard draft" http://www.mpirt.org, (9/1998).

[5] Argonne National Laboratory and Mississippi State University's MPI implementation, "MPICH": http://www.mcs.anl.gov//mpi/index.html .

[6] American National Standard for Information Systems, "Fiber-distributed data interface (FDDI) - Token Ring Media Access Control (MAC)" ANSI X3.139-1987, (7/1987). American National Standard Institute.

[7] Qin Zheng, "Synchronous Bandwidth Allocation in FDDI Networks", IEEE Transactions on Parallel And Distributed Systems, Vol 6, No 12, (12/1993).

[8] S.Pakin, M.Lauira, and A.Chien, "High Performance Messaging on Workstations: Illinois Fast Message (FM) for Myrinet". In Supercomputing, (12/95). Available: http: //www.csag.cs.uiuc.edu/papers/myrinet-fm-sc95.ps

[9] A.Bilas and E.W.Felten ShrimpRPC, "Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface", Journal Of Parallel And Distributed Computing 40, pp. 138-146 (1997).

[10] Jehoshua Bruck et al, "Efficient Message Passing (MPI) for Parallel Computing on Clusters of Workstations", Journal Of Parallel And Distributed Computing, 40, pp. 19-34 (1997).

[11] V.Karamcheti and A.Chien, "Software Overhead in Messaging Layers: Where does the time go?" In proceeding of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), (1994). Available http://www.csag.cs.uiuc.edu/papers/asplos94.ps

[12] Sung-Yong Park, Salim Hariri, "A High Performance Message-Passing System for Network of Workstations", The Journal of Supercomputing, 11, pp. 159-179 (1997).

[13] Prasenjit Sarkar, "Adapting the Network Interface for High-Performance Computing: CNI Approach", Journal of Supercomputing 11, pp. 181-200(1997).

[14] Nayeem Islam, "Customized Message Passing", Journal of Parallel and Distributed Computing 41, pp. 205-224(1997).

[15] Ian Foster, Carl Kesselman, Steve Tuecke, "The Nexus Approach to Integrating Multithreading and Communication", Journal of Parallel and Distributed Computing 37, pp. 70-82(1996).

[16] Unix Systems laboratories, "Data Link Provider Interface Specification (DLPI)", Unix International Incorporated (10/1998)

[17] Ohio Computing Center, " MPI primer/developing with LAM", Ohio Computing Center (12/95)

[18] Butler, R., and Lusk, " Monitors, messages and clusters: The P4 parallel programming system", Parallel computing, pp. 547-564(04/1994).

[19] Walter J. Goralski, " Introduction to ATM networking ", McGraw-Hill Series on Computer Communications, 1995, book,
ISBN: 0-07-024043-4

[20] William Gropp, Ewing Lust, " An Abstract Device Definition to Support the Implementation of a High-Level Point-to-Point" Message Passing Interface", Mathematics and Computer Science Division, Argonne National Laboratory, PrePrint MCS-P392-1993 (3/1995)

[21] Paul Ferguson, Geoff Huston, " Delivering QoS on the Internet and in Corporate Networks", Wiley Computer Publishing, (1998), book,
ISBN: 0-471-24356-2

[22] Sun Microsystems Inc, "ts_dptbl(4) manual page. SunOS 5.7 manual. Section 4.

[23] P.G.Sobalvarro, " Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors ", Phd Thesis, Department of Electrical Engineering and Computer Science MIT, (1997).

[24] D.K.Panda, L.M.Ni, "Special Issue on Workstation Clusters and Network-Based Computing", Journal of Parallel and Distributed Computing 40, pp. 1-3 (1997).

[25] G.A Geist,J.A. Kohl, P.M. Papadopoulos "PVM and MPI: A Comparison of Features", unpublished article, (1996).

[26] Krishnan R. Subramaniam, " A Communication Library Using Active Messages to improve Performance of PVM", Journal of Parallel and Distributed Computing 37, pp. 146-152 (1996).

[27] Von Eicken, Thorsten Culler, David E., Golstein and others, " Active Messages: A Mechanism for Integrated Communication and Computation.", Proceedings of the 19th International Symposium of Computer Architecture (1992).

[28] T. M. Anderson and R.S. Cornelius. " High performance switching with Fiber Channel", In Digest of Papers Compcon 1992, pp 261-268, IEEE Computer Society Press, 1992. Los Alamitos, Calif.

[29] William Group, Ewing Lust, "An Abstract Device Definition to Support the Implementation of a High-Level Point-to-Point Message-Passing Interface", PREPRINT MCS-P392-1193, Mathematics and Computer Science Division, Argonne Laboratory.

[30] William Group, Ewing Lusk, "Creating a New MPICH Device using the Channel Interface", Technical Memorandum No. 213, ANL/MCS-TM-213, Mathematics and Computer Science Division, Argonne Laboratory.

[31] P. Druschel, and L. L. Peterson, "Fbufs, A high-bandwidth cross-domain transfer facility", Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP), pp. 189-202, Asheville, North Carolina, December 1993. ACM SIGOPS, ACM Press. Available from ftp://ftp.cs.arizona.edu/xkernel/Papers/fbuf.ps

[32] H.Chu, "Zero-copy Solaris", In Proceedings of the USENIX Annual Technical Conference, pp. 253-264, San Diego, California, January 1996. Available from http://www.usenix.org

[33] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: User-level network interface for parallel and distributed computing, Proceedings of the 15th ACM and Symposium on Operating Systems Principles, pages 40-53, December 1995. Available from http://www2.cs.cornell.edu/U-Net/papers/sosp.pdf

[34] S.Rodriques, T. Anderson, and D. Culler, "High Performance local-area communication using Fast Socket ", In Proceedings of the USENIX 1997 Technical

Conference, pp. 253-264, San Diego, California, January 1996. Available from http://now.cs.berkeley.edu/Papers2

[35] Lawrence S.Brakmo, Larry L. Peterson, "Performance Problems in BSD4.4 TCP", Department of Computer Science, University of Arizona, Tucson AZ 85721

[36] Protocol Engines Incorporated, "XTP Protocol Definition Revision 3.6", Procotol Engines, 01/11/92

[37] Berny Goodheart & James Cox, "The Internals of UNIX System V Release 4: Open System Design", Prentice Hall, ISBN 013 098138 9 1994

[38] W. Richard Stevens, "UNIX Network Programming Volume I Networking APIs: Sockets and XTI", Prentice Hall, ISBN 0-13-490012-X

Appendix

MPI-GB ADI interface header file

```
#ifndef _DMCH_INCLUDED
#define _DMCH_INCLUDED

#define MPID_NOT_HETERO

/* Whether an MPID_Check_incoming should block or not */
typedef enum { MPID_NOTBLOCKING = 0, MPID_BLOCKING }
MPID_BLOCKING_TYPE;

/*************************************************************************
  MPID Send and Receive Handle
*************************************************************************/

typedef struct {
  int    is_non_blocking;
  void   *start;
  int     bytes_as_contig;
  /* used by ADI */
  int     status;
  int     recv_handle;
  int     recv_addr;
  int     recv_length;
} MPID_SHANDLE;

typedef struct {
  int    is_non_blocking;
  void   *start;
  int     bytes_as_contig;
  /* used by ADI */
  int     is_tmp;
  int     sender;
  int     send_handle;
} MPID_RHANDLE;


#define MPID_Alloc_send_handle( ctx, a )
#define MPID_Alloc_recv_handle( ctx, a )
#define MPID_Free_send_handle( ctx, a )
#define MPID_Free_recv_handle( ctx, a )  {if ((a)->is_tmp == 1) \
                                  free((a)->start); \
                                  (a)->is_tmp = 0; \
                                  (a)->start=NULL;}
#define MPID_Reuse_send_handle( ctx, a )
#define MPID_Reuse_recv_handle( ctx, a ) {if ((a)->is_tmp == 1) \
                                  free((a)->start); \
                                    (a)->is_tmp = 0; \
                                  (a)->start = NULL; }
#define MPID_Set_send_is_nonblocking( ctx, a, v ) (a)->is_non_blocking
= v
```

```
#define MPID_Set_recv_is_nonblocking( ctx, a, v ) (a)->is_non_blocking
= v


/*****************************************************************
****
   Contact with the device layer is made here.

*****************************************************************
****/
#define MPID_Post_send(ctx,dmpi_send_handle) \
           MPID_GB_Post_send(dmpi_send_handle)
#define MPID_Post_send_ready(ctx,dmpi_send_handle) \
           MPID_GB_Post_send_ready(dmpi_send_handle)
#define MPID_Post_send_sync(ctx,dmpi_send_handle)  \
           MPID_GB_Post_send_sync(dmpi_send_handle)
#define MPID_Complete_send(ctx,dmpi_send_handle)   \
           MPID_GB_Complete_send(dmpi_send_handle)
#define MPID_Test_send( ctx, dmpi_send_handle ) \
           ((volatile)((dmpi_send_handle)->completer == 0) ? 1 : 0)

#define MPID_Post_recv(ctx,dmpi_recv_handle ) \
           MPID_GB_Post_recv(dmpi_recv_handle)
#define MPID_Complete_recv(ctx,dmpi_recv_handle) \
           MPID_GB_Complete_recv(dmpi_recv_handle)
#define MPID_Test_recv( ctx, dmpi_recv_handle ) \
           ((volatile)((dmpi_recv_handle)->completer == 0) ? 1 : 0)


#define MPID_Ctx( request )                (request)->chandle.comm-
>ADIctx
#define MPID_Test_request( ctx, request ) \
            ( (request)->chandle.handle_type == MPIR_SEND ? \
            MPID_Test_send(ctx,&(request)->shandle) : \
            MPID_Test_recv(ctx,&(request)->rhandle))
#define MPID_Clr_completed( ctx, request )   (request)-
>chandle.completer = 1
#define MPID_Set_completed( ctx, request )   (request)-
>chandle.completer = 0

#define MPID_Check_device(ctx,blocking)
MPID_GB_Check_device(blocking)
#define MPID_Iprobe( ctx, tag, source, context_id, flag, status ) \
            MPID_GB_Iprobe( tag, source, context_id, flag, status )
#define MPID_Probe( ctx, tag, source, context_id, status ) \
            MPID_GB_Probe( tag, source, context_id, status )

#define MPID_INIT(argc,argv)             MPID_GB_Init( argc, argv )
#define MPID_END(ctx)                    MPID_GB_End()
#define MPID_ABORT( ctx, errorcode )     MPID_GB_Abort( errorcode );
#define MPID_CANCEL( ctx, r )

#define MPID_Myrank( ctx, rank ) MPID_GB_Myrank( rank )
#define MPID_Mysize( ctx, size ) MPID_GB_Mysize( size )

/*****************************************************************
**********
```

```
  Point-to-Point Extension Routines
*****************************************************************
*********/

#define MPID_Blocking_recv(ctx,dmpi_recv_handle ) \
          MPID_GB_Blocking_recv(dmpi_recv_handle)
#define MPID_Blocking_send(ctx, dmpi_send_handle)  \
          MPID_GB_Blocking_send(dmpi_send_handle)
#define MPID_Blocking_send_ready(ctx, dmpi_send_handle) \
          MPID_GB_Blocking_send_ready(dmpi_send_handle)

/*****************************************************************
**********
  Environment Extension Routines
*****************************************************************
*********/

#define MPID_NODE_NGBE( ctx, name, len ) MPID_GB_Node_name( name, len )
#define MPID_Version_name( ctx, name )   MPID_GB_Version_name( name )
#define MPID_WTIME(ctx)                  MPID_GB_Wtime()
#define MPID_WTICK(ctx)                  MPID_GB_Wtick()

/*****************************************************************
********
  Collective Extension Routines
*****************************************************************
*********/

#ifdef MPID_USE_ADI_COLLECTIVE

#define MPID_Comm_init(ctx,comm,newcomm)
MPID_GB_Comm_init(comm,newcomm)
#define MPID_Comm_free(ctx,comm) MPID_GB_Comm_free(comm)

#define MPID_Barrier(ctx,comm) MPID_GB_Barrier(comm)
#define MPID_Bcast(ctx,comm) MPID_GB_Bcast(comm)

#else /* No MPID_USE_ADI_COLLECTIVE is defined */

#define MPID_Comm_init(ctx,comm,newcomm) MPI_SUCCESS
#define MPID_Comm_free(ctx,comm)         MPI_SUCCESS

#endif /* MPID_USE_ADI_COLLECTIVE */

/*
   Context and Communicator operations
 */

/*****************************************************************
**********
  MPID Packets Size
  - can be set by user if the following defined
*****************************************************************
*********/
#ifdef MPID_PKT_VAR_SIZE
#undef MPID_PKT_VAR_SIZE /* Not yet implemented */
#endif
```

```
#ifdef MPID_PKT_VAR_SIZE
#define MPID_SetPktSize(len)  MPID_GB_SetPktSize(len)
#else
#define MPID_SetPktSize(len)
#endif /* MPID_PKT_VAR_SIZE */

/*****************************************************************************
**********
  MPID Debugging routines
*****************************************************************************
*********/
#ifdef MPID_HAS_DEBUG
#undef MPID_HAS_DEBUG
#endif

#ifdef MPID_HAS_DEBUG

#define MPID_SetDebugFlag(ctx,flag)     MPID_GB_SetDebugFlag(flag)
#define MPID_SetDebugFile(filename)     MPID_GB_SetDebugFile(filename)
#define MPID_SetSpaceDebugFlag(flag)    MPID_GB_SetSpaceDebugFlag(flag)
#define MPID_SetMsgDebugFlag(ctx,flag) MPID_GB_SetMsgDebugFlag(flag)
#define MPID_Set_tracefile(filename)    MPID_GB_Set_tracefile(filename)

#endif /* MPID_HAS_DEBUG */

/*****************************************************************************
*********
   This device prefers that the data be prepacked (at least for now)
*****************************************************************************
*********/

#define MPID_PACK_IN_ADVANCE
#define MPID_RETURN_PACKED

/*****************************************************************************
**********
  Device-only information
  For ADI compilation use
*****************************************************************************
*********/

#include "mpiimpl.h"

#ifdef HAS_XDR
#undef HAS_XDR
#endif

#endif /* DMCH_INCLUDED */
```

Performance Analysis Test codes

Roundtrip Latency

```
/*
--    Roundtrip (Ping/Pong) latency.c  -- testing the ping/pong latency
of MPI-GB
*/

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include "mpi.h"
#include "gb.h"


#define max_data_size (64 * 1024)
#define max_num_reps max_data_size
#define max_num_sets 256

#define SEND_TAG  9


int max_nodes, node_id, verbose;
int *send_data, *recv_data;
int data_size, num_reps, num_sets, total_messages;
int *cmd_line;
double elapsed_time[max_num_sets], overhead_time[max_num_sets];
double start_time, stop_time, ave_time, median_time,
ave_overhead,min_time,max_time;
MPI_Datatype send_type, recv_type;
MPI_Status status;


void show_time_for_set(double time) {

if(verbose) {
      printf("Elapsed time = %.3f ms \n", (float) time * 1000.);
      printf("Data rate = %3f MB/s \n", sizeof(int) * num_reps *
data_size / (time * 1048576.0));
  }

printf(" Average latency of 1 %d-byte message = %.3f microseconds. \n
\n",
data_size* sizeof(int), (float) time* 1000000.0 / (float) num_reps/2);

}

void show_things_for_set(double mintime,double avetime,double
maxtime,double medi_time) {
```

```
printf(" 1 %d-byte P.P AL= %.3f MinL= %.3f MaxL= %.3f MedL=  %.3f
\n",data_size* sizeof(int) , (float) avetime* 1000000.0 / (float)
num_reps/2,
(float) mintime* 1000000.0 / (float) num_reps/2,
(float) maxtime* 1000000.0 / (float) num_reps/2,
(float) medi_time* 1000000.0 / (float) num_reps/2);

}

void node0() {

int i,j;
double swap;

/* initialize data */

for(i=0;i < data_size ; i++)
  send_data[i] = i;

if(verbose) {
printf("Ping-ponging %i %i-byte messages %i times. \n", num_reps,
data_size * sizeof(int), num_sets);
printf("The first trial will be ignored in result summary \n \n");
}

fflush(stdout);

for(i=0; i< num_sets; i++) {

start_time = MPI_Wtime();

for( j=0 ; j < num_reps; j++) {

 MPI_Send(send_data, data_size, send_type,1, SEND_TAG, MPI_COMM_WORLD)

 MPI_Recv(recv_data, data_size, recv_type,1, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);

 }

 stop_time = MPI_Wtime();

 elapsed_time[i] = stop_time - start_time;

 }

/* Calculate the timer overhead */

ave_overhead = 0;

for( i =0 ;i< num_sets; i++)
{
 start_time = MPI_Wtime();
 stop_time=MPI_Wtime();
 overhead_time[i]=stop_time-start_time;
 ave_overhead +=overhead_time[i];
}
```

```
ave_overhead /= num_sets;

if(verbose) {
 printf("Timer overhead amounted to :\n");
 show_time_for_set(ave_overhead);
 printf("Timer granularity: %.3f milliseconds\n\n", (float)
MPI_Wtick()*1000.0); printf("Timer overhead is being substracted \n");
 }

/* Sort the results (buble sort) */
/* the first set is ignored (elapsed_time[0] is not used  */

for(i = num_sets - 1; i > 1; i--) {

  for(j = 1; j < i ; j ++) {

  if (elapsed_time[j] > elapsed_time[j+1]) {
     swap = elapsed_time[j];
     elapsed_time[j] = elapsed_time[j+1];
     elapsed_time[j+1] = swap;
    }

  }
}

  /* Substract overhead; calculate the ave time */

  ave_time = 0 ;
  for(i = 1 ; i < num_sets; i++) {
    elapsed_time[i] -= ave_overhead;
    ave_time +=elapsed_time[i];
  }


  ave_time /=(num_sets - 1);

   median_time = (elapsed_time[num_sets/2] + elapsed_time[(num_sets +
1)/2])/2;
  if(verbose) {
   printf("\n Minummum time \n:");
   show_time_for_set(elapsed_time[1]);
   printf("\n Avarege over all but the first trial:\n");
   show_time_for_set(ave_time);
   median_time = (elapsed_time[num_sets/2] + elapsed_time[(num_sets +
1)/2])/2;
   printf("\n Median Time \n");
   show_time_for_set(median_time);
   }

  min_time = elapsed_time[1];
  max_time =  elapsed_time[num_sets -1];

  show_things_for_set(min_time,ave_time,max_time,median_time);
}

void node1() {
```

```
int i;

/* initialize data */

for(i=0;i < data_size ; i++)
  send_data[i] = -i;


      fflush(stdout);

for(i=0; i< total_messages ; i++) {

MPI_Recv(recv_data, data_size, recv_type, 0, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);

MPI_Send(send_data, data_size, send_type,0, SEND_TAG, MPI_COMM_WORLD);
}
}

void command_line (int argc, char *argv[], int *num_sets, int
*num_reps, int *data_size, int *verbose) {

*num_sets=5;
*num_reps=20;
*data_size=16;
*verbose=0;

if(argc > 1) {
  *num_sets = atoi(argv[1]);
  if(*num_sets > max_num_sets) {
  printf("Too many sets (%d vs %d). \n", *num_sets, max_num_sets);
  exit(-1);
  }
}

if(argc > 2) {
  *num_reps = atoi(argv[2]);
  if(*num_reps > max_num_reps) {
  printf("Too many sets (%d vs %d). \n", *num_sets, max_num_sets);
  exit(-1);
  }
}

if (argc > 3) {
  *data_size=atoi(argv[3])/4;
  if(*data_size > max_data_size)
   {
     printf("To much data (%d vs %d ) \n", *data_size, max_data_size);
     exit(-1);
    }
 }

if(argc > 4)
   *verbose = atoi(argv[4]);

}
```

```
int main (int argc, char *argv[]) {

/* Allocate send and receive buffers */

send_data = malloc(max_data_size * sizeof(int));
recv_data = malloc(max_data_size * sizeof(int));

printf("Before the init \n");
MPI_Init(&argc, &argv);
printf("After the init \n");
MPI_Comm_size(MPI_COMM_WORLD, &max_nodes);
MPI_Comm_rank(MPI_COMM_WORLD, &node_id);

if(max_nodes > 2) {
  printf("Error[%i] : test not setup for more than two nodes \n",
node_id);
  exit(-1);
}

send_type = MPI_INT;
recv_type = MPI_INT;

cmd_line = malloc(sizeof(double) * 8);

if(node_id == 0) {
   command_line(argc, argv, &num_sets, &num_reps, &data_size,
&verbose);
   cmd_line[0] = num_sets;

   cmd_line[2] = num_reps;
   cmd_line[4] = data_size;
   cmd_line[6] = verbose;

   MPI_Send(&cmd_line[0], 1 , send_type, 1, SEND_TAG, MPI_COMM_WORLD);

   MPI_Send(&cmd_line[2], 1 , send_type, 1, SEND_TAG, MPI_COMM_WORLD);

   MPI_Send(&cmd_line[4], 1 , send_type, 1, SEND_TAG, MPI_COMM_WORLD);

   MPI_Send(&cmd_line[6], 1 , send_type, 1, SEND_TAG, MPI_COMM_WORLD);

   total_messages = num_sets * num_reps;

   node0();
 }
 else {

   send_type = MPI_INT;
   recv_type = MPI_INT;

   MPI_Recv(&cmd_line[0],1,recv_type, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);


   MPI_Recv(&cmd_line[2],1,recv_type, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
```

```
   MPI_Recv(&cmd_line[4],1,recv_type, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);


   MPI_Recv(&cmd_line[6],1,recv_type, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);

   num_sets = cmd_line[0];
   num_reps = cmd_line[2];
   data_size = cmd_line[4];
   verbose = cmd_line[6];

   total_messages=num_sets * num_reps;
   node1();
  }

 MPI_Finalize();

}
```

Bandwidth

```
/*
--    bandwidth.c  -- testing the bandwidth of MPI
*/

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include "mpi.h"
#include "gb.h"


#define max_data_size (64 * 1024)
#define max_num_reps max_data_size
#define max_num_sets 256

#define SEND_TAG  9


int max_nodes, node_id, verbose;
int *send_data, *recv_data;
int data_size, num_reps, num_sets, total_messages;
int *cmd_line;
double start_time, stop_time, ave_time, median_time, ave_overhead;
double elapsed_time[max_num_sets], overhead_time[max_num_sets];
MPI_Datatype send_type, recv_type;
MPI_Status *status;
MPI_Request *request;


void show_time_for_set(double time) {

if(verbose) {
      printf("Elapsed time = %.3f ms \n", (float) time * 1000.0);
      printf("Avarage latency of 1 message = %.3f  microseconds. \n\n",
(float) time * 1000000 / (float) num_reps );
  }

printf(" Data rate of %d - byte messages = %.3f MB/s \n\n", data_size *
sizeof(int), sizeof(int) * num_reps * data_size / (time * 1048576.0));

}

void show_things_for_set(double avetime)
{

printf(" Data rate of %d - byte messages = %.3f MB/s lat=%.3f ms \n\n",
data_size * sizeof(int), sizeof(int) * num_reps * data_size / (avetime
* 1048576.0), (float) avetime * 1000000 / (float) num_reps );

}

void node0() {

int i,j;
```

```
double swap;

/* initialize data */

for(i=0;i < data_size ; i++)
  send_data[i] = i;

if(verbose)
   printf("Sendinf %i %i-byte messages %i times. \n", num_reps,
data_size * sizeof(int), num_sets);

if(verbose) {
printf("The first trial will be ignored in result summary \n \n");
}

fflush(stdout);

for(i=0; i< num_sets; i++) {

start_time = MPI_Wtime();

for( j=0 ; j < num_reps; j++) {

 MPI_Send(send_data, data_size, send_type,1, SEND_TAG, MPI_COMM_WORLD);
}

 MPI_Recv(recv_data, data_size, recv_type,1, MPI_ANY_TAG,
MPI_COMM_WORLD, status);

 stop_time = MPI_Wtime();

 elapsed_time[i] = stop_time - start_time;

 }

/* Calculate the timer overhead */

ave_overhead = 0;
for( i =0 ;i< num_sets; i++)
{
 start_time = MPI_Wtime();
 stop_time=MPI_Wtime();
 overhead_time[i]=stop_time-start_time;
 ave_overhead +=overhead_time[i];
}

ave_overhead /= num_sets;

if(verbose) {
 printf("Timer overhead amounted to :\n");
 show_time_for_set(ave_overhead);
 printf("Timer granularity: %.3f milliseconds\n\n", (float)
MPI_Wtick()*1000.0); printf("Timer overhead is being substracted \n");
 }

/* Sort the results */
/* the first set is ignored (elapsed_time[0]) is not used  */
```

```
for(i = num_sets - 1; i > 1; i--) {

  for(j = 1; j < i ; j ++) {

  if (elapsed_time[j] > elapsed_time[j+1]) {
     swap = elapsed_time[j];
     elapsed_time[j] = elapsed_time[j+1];
     elapsed_time[j+1] = swap;
    }

  }
}

  /* Substract overhead; calculate the ave time */

  ave_time = 0 ;
  for(i = 1 ; i < num_sets; i++) {
    elapsed_time[i] -= ave_overhead;
    ave_time +=elapsed_time[i];
  }

  ave_time /=(num_sets   1);

  if(verbose) {
    printf("\n Minummum time \n:");
    show_time_for_set(elapsed_time[1]);
    printf("\n Avarege over all but the first trial:\n");
    show_time_for_set(ave_time);
    median_time = (elapsed_time[num_sets/2] + elapsed_time[(num_sets +
1)/2])/2;
    printf("\n Median Time \n");
    show_time_for_set(median_time);
   }

show_things_for_set(ave_time);
}



void node1() {
int i,j;

/* initialize data */

for(i=0;i < data_size ; i++)
  send_data[i] = -i;

if (verbose) {

      printf("Sending  %i %i-byte messages %i times. \n \n", num_reps,
data_size * sizeof(int), num_sets);

      fflush(stdout);
          }

for(i=0; i< num_sets ; i++) {
```

```
for(j=0; j< num_reps ; j++) {


MPI_Irecv(recv_data, data_size, recv_type, 0, MPI_ANY_TAG,
MPI_COMM_WORLD, &request[j]);

}

MPI_Waitall(num_reps,request, status);

MPI_Send(send_data, data_size, send_type,0, SEND_TAG, MPI_COMM_WORLD);
}
}

void command_line (int argc, char *argv[], int *num_sets, int
*num_reps, int *data_size, int *verbose) {

*num_sets=5;
*num_reps=20;
*data_size=16;
*verbose=0;

if(argc > 1) {
  *num_sets = atoi(argv[1]);
  if(*num_sets > max_num_sets) {
  printf("Too many sets (%d vs %d). \n", *num_sets, max_num_sets);
  exit(-1);
  }
}

if(argc > 2) {
  *num_reps = atoi(argv[1]);
  if(*num_reps > max_num_reps) {
  printf("Too many sets (%d vs %d). \n", *num_sets, max_num_sets);
  exit(-1);
  }
}

if (argc > 3) {
  *data_size=atoi(argv[3])/4;
  if(*data_size > max_data_size)
   {
     printf("To much data (%d vs %d ) \n", *data_size, max_data_size);
     exit(-1);
    }
 }

if(argc > 4)
   *verbose = atoi(argv[4]);
}

int main (int argc, char *argv[]) {

/* Allocate send and receive buffers */

send_data = malloc(max_data_size * sizeof(int));
recv_data = malloc(max_data_size * sizeof(int));
```

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &max_nodes);
MPI_Comm_rank(MPI_COMM_WORLD, &node_id);

if(max_nodes > 2) {
  printf("Error[%i] : test not setup for more than two nodes \n",
node_id);
  exit(-1);
}

send_type = MPI_INT;
recv_type = MPI_INT;

cmd_line = malloc(sizeof(double) * 8);

command_line(argc, argv, &num_sets, &num_reps, &data_size, &verbose);
total_messages = num_sets * num_reps;

status = (MPI_Status *) malloc (num_reps * sizeof(MPI_Status));
request = (MPI_Request *) malloc (num_reps * sizeof(MPI_Request));

if(node_id == 0) {
   node0();
 }
 else {
   node1();
  }

 MPI_Finalize();

}
```

Percentage of missing deadlines packets.

Sender

```
#include <stdio.h>              /* For fprintf, perror */
#include <stdlib.h>             /* For exit */
#include <sys/types.h>          /* For socket */
#include <unistd.h>
#include <string.h>             /* For bzero */
#include <signal.h>             /* For sigemptyset, sigaddset */
#include <sys/time.h>           /* For setitimer */
#include <sys/stropts.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <memory.h>
#include <sched.h>

#include "GB.h"

#define INTERVAL 100E3 /* Send interval in us (100 ms) */

int main(int ac, char *av[])
{
  unsigned short         tag = 0;
  struct   itimerval     itimer = {{ 0, INTERVAL}, { 0, INTERVAL}};
  sigset_t               signalSet;
  int     com_id;
  int     size;
  int  rc;
  int success;
  char *packet;
  struct sched_param schd;
  GB_resource_struct test_res;



 /* All receivers and transmitters are assinged with unique com id here
*/
  /* IP address read from configuration file */

  com_id = GB_Init();

  if(com_id <= 0)
   {
   fprintf(stderr, "GB protocol unable to initialize");
   exit(1);
   }

  test_res.rlength = 0;
  test_res.pay_type= 1;   /* Synchronous packets */
  test_res.payload = 157; /* Synchronous units  */
  test_res.slength = 4000;

  success = GB_Commit(&test_res);
```

```
  if(success < 0)
   {
   fprintf(stderr, "GB protocol unable to commit");
   exit(1);
   }

   packet = test_res.send_buffers;

/* Initialize signal set to wait for SIGALRM or SIGINT */
  if (sigemptyset(&signalSet) < 0) {
    perror("sigemptyset");
    exit(1);
  }
  if (sigaddset(&signalSet, SIGALRM) < 0) {
    perror("sigaddset: SIGALRM");
    exit(1);
  }
  if (sigaddset(&signalSet, SIGINT) < 0) {
    perror("sigaddset: SIGINT");
    exit(1);
  }

  /* Block signals so sigwait will work */
  if (sigprocmask(SIG_BLOCK, &signalSet, NULL) != 0) {
    perror("sigprocmask");
    exit(1);
  }

  /* Set timer */
  if (setitimer(ITIMER_REAL, &itimer, NULL) < 0) {
    perror("setitimer");
    exit(1);
  }



/* Wait for alarm signals and send UDP messages */
  for (;;) {
    int signal;
    if ((sigwait(&signalSet,&signal) == 0) && (signal == SIGALRM))
      if (GB_Send(com_id, packet, test_res.slength ) < 0)
      {
        perror("GB_send");
        exit(1);
      }

     *(u_short *)(packet+22) = tag++;

    }
    else if (signal == SIGINT)
      exit(0);
    else {
      perror("sigwait");
      exit(1);
    }
 }
  return 0;}
```

Receiver

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/bufmod.h>
#include <sys/signal.h>
#include <sys/stream.h>
#include <sys/systeminfo.h>

#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stropts.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sched.h>

#include "GB.h"


#define INTERVAL  100E6 /* Expected send interval in nanoseconds (100
ms) */
#define TOLERANCE  5E6 /* +/- acceptable timing tolerance in
nanoseconds (5 ms) */

int main(int argc, char *argv[])
{
  int com_id, rc;
  int success;
  char *packet;
  struct sched_param schd;
  GB_resource_struct test_res;


  /* Set minimum realtime priority */
  if ((schd.sched_priority = sched_get_priority_min(SCHED_FIFO)) == -1)
{
    perror("sched_get_priority_min");
    exit(1);
  }
  if (sched_setscheduler(0, SCHED_FIFO, &schd) == -1)
    perror("\aWarning: Not a realtime process");


  /* All receivers and transmitters are assinged with unique com id
here */
  /* IP address read from configuration file */
```

```
com_id = GB_Init();

if(com_id <= 0)
 {
 fprintf(stderr, "GB protocol unable to initialize");
 exit(1);
 }

test_res.rlength = 4000;
test_res.pay_type= 1;    /* Sync. packets */
test_res.payload =  157; /* Synchronous units */
test_res.slength = 0;

success = GB_Commit(&test_res);

if(success < 0)
 {
 fprintf(stderr, "GB protocol unable to commit");
 exit(1);
 }

packet = test_res.rec_buffers;


/* Wait for packets, check tag value for missing ones,
 * & calculate message intervals */


for (;;) {
   hrtime_t              current_time, last_time;
   long                  rate;
   unsigned short        current_tag;
   static unsigned short last_tag = 0;

   /* Wait for packets */
   rc = GB_Poll(com_id,test_res.rlength);

   /* Got a packet */
   if (rc > 0) {

     /* Calculate transmission rate */
     current_time = gethrtime();
     rate = current_time - last_time;

     /* Notify if out of range */
     if ((rate > INTERVAL + TOLERANCE) || (rate < INTERVAL -
TOLERANCE))
       printf("Packet: %d; Rate: %d\n", current_tag, (int)rate);

     last_time = current_time;

     /* Check for missing packets */
     current_tag = *(unsigned short *) (packet + 22);
     if (++last_tag != current_tag) {
       printf("Packet missing: last %d; current %d\n",
            last_tag, current_tag);
       last_tag = current_tag;
```

```
    }
  }
  else {
    perror("Packet read error");
    exit(1);
  }
```